WL-TR-91-1102

Suitability of Ada for Real-Time Model Based Vision Applications

Robert Kaplan and Phil Hanselman
Data & Signal Processing Group
Systems Avionics Division

November 1991

Final Report for Period July 90 to July 91

AVIONICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-6543

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

Robert A. Kaplan

ROBERT A. KAPLAN
Data & Signal Processing Group
Information Processing
Technology Branch
Systems Avionics Division

Edward L. Gliatti

EDWARD L. GLIATTI, Chief
Information Processing
Technology Branch
Systems Avionics Division

FOR THE COMMANDER

Charles H. Krueger

CHARLES H. KRUEGER, Chief
System Avionics Division
Avionics Directorate

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify WL/AAAT, WPAFB, OH 45433-6543 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	November 1991	July 1990 - July 1991 Final

4. TITLE AND SUBTITLE

Suitability of Ada for Real-Time Model Based Vision Applications

5. FUNDING NUMBERS

PE: 63109F
PN: 2273
TN: 01
WU: 04

6. AUTHOR(S)

Robert A. Kaplan and Phillip B. Hanselman

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Information Processing Technology Branch (WL/AAAT)
Data & Signal Processing Group
Systems Avionics Division
Wright-Patterson AFB OH 45433-6543

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Robert A. Kaplan (513) 255-7701
Avionics Directorate (WL/AAAT)
Air Force Systems Command
Wright-Patterson AFB OH 45433-6543

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

WL-TR-91-1102

11. SUPPLEMENTARY NOTES

The computer software contained herein is theoretical and/or references that in no way reflect Air Force-owned or-developed computer software

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release;
distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The purpose of this study was to establish a baseline for assessing the execution efficiency of Model Based Vision (MBV) algorithms coded in both the Ada and C programming languages. To facilitate this study a key algorithm used in The Analytic Science Corporation's (TASC) laboratory MBV system called Fast Pairwise Nearest Neighbor (FASTPNN) was recoded from its orignal C form to Ada, and benchmarked on both a VAX 11/780 and MIPS MAGNUM 3000 computer system. Comparisions of C execution efficiency versus Ada execution efficiency as well as MIPS MAGNUM 3000 execution efficiency versus VAX 11/780 execution efficiency were made. The benchmark results indicate that when Ada run-time checks are suppressed, Ada and C are roughly equivalent in terms of inherent execution efficiency. Differences between Ada and C execution efficiency can be attributed to deviations in a particular compiler's maturity. Ada run-time checks impose between a 43% and 65% execution penalty when compared with Ada executing with all run-time checks suppressed. Depending on whether C or Ada was the language of interest, the MIPS MAGNUM 3000 demonstrated between a 9 (Ada) to 22 (C) times throughput advantage over the VAX 11/780.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Ada, C, Real-Time, Model-Based Vision, Fast Pairwise Nearest Neighbor Algorithm, Benchmark	143
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std Z39-18

TABLE OF CONTENTS

TABLE OF CONTENTS (CONTINUED)

FIGURES

TABLES

1. INTRODUCTION. The purpose of this study was to establish a baseline for assessing the execution efficiency of Model Based Vision (MBV) algorithms coded in both the Ada and C programming languages. Model-based vision systems compare predictions of target signature in sensed data with information extracted from sensed data to achieve target recognition. To date, prediction of electromagnetic signature for Synthetic Aperture Radar (SAR) is the emphasized sensed data. Under contract with Wright Laboratory, The Analytic Science Corporation (TASC) has extracted algorithms from their MBV system and delivered them to the Government. The algorithms delivered to the Government are to be used as benchmarks for evalutating the performance of different processors for MBV applications. All of the algorithms delivered are coded in the C programming language.

To evaluate the suitability of Ada for MBV applications, an important MBV algorithm from the TASC MBV system, called Fast Pairwise Nearest Neighbor (FASTPNN), was translated in its orignal C form to Ada and benchmarked on both a VAX 11/780 and MIPS Magnum 3000 computer. While the FASTPNN algorithm comprises only a small subset of the entire TASC MBV system, in terms of the general types of processing requirements it imposes, it is representative of a significant portion of the TASC MBV system.

2. DESCRIPTION OF PAIRWISE NEAREST NEIGHBOR ALGORITHM.

The Pairwise Nearest Neighbor (PNN) algorithm is a vector quantization procedure used inside the information extraction portion of the TASC MBV system. The PNN algorithm is used to represent the significant characterists of a large number of image samples (or vectors) with a smaller specified number of vectors such that the derived set of vectors represent the original set of vectors as "best" as possible with respect to coordinate position and weight (gray level). The PNN algorithm derives a specified number of quantization vectors by progressively merging together pairs of vectors with minimal weighted distance between their centroids.

The TASC MBV system uses a vector quantizer which consists of the FASTPNN vector quantizer cascaded with a Linde-Buzo-Gray (LBG) vector quantizer. The LBG vector quantizer is an iterative algorithm that produces a set of quantization vectors that minimize mean square quantization error; however the algorithm tends to converge to local minima unless good estimates for initial quantization vectors are available. The PNN vector quantizer provides a near optimal set of quantization vectors, that support the functional needs of the LBG algorithm.

2.1 Full Search Implementation of PNN Algorithm

The key to quick execution of the PNN algorithm is quickly finding the closest pairs of entries (vectors) among the distributed set. Figure 1 describes the sequence of operations in the full search implementation of PNN. Figure 1 shows that the full search implementation of PNN is an iterative procedure where on each iteration the pair of vectors which generates the least error when merged (the closest pair) is merged. The algorithm explicitly requires that, at each iteration, each vector's nearest neighbor be found. The calculation to determine the closest vector pair is essentially a weighted distance calculation where the distance between the two vectors are weighted by the gray level (or weight) of each of the two vectors. The new vector is chosen to minimize the error incurred by replacing the two original vectors into one new vector.

In general, the number of distance calculations to find the two closest vectors among a randomly distributed set is given by

$$DIST\_CALC = (N * (N-1)) / 2 \qquad (1)$$

where DIST_CALC is the number of permutations of distance calculations required and N is the number of entries (vectors). Thus, if one were to intitiate a full search PNN with 1000 entries, then the first iteration would require 499,500 distance calculations to find the closest vector pair for merging. Once the first iteration was completed, the following iterations would require much fewer distance calculations to find the closest vector pairs since only the distances from the "new" vector to each of the remaining entries would need to be recalculated. The distance calculations among the remaining entries would not need to be recalculated. Thus, in our example, only 999

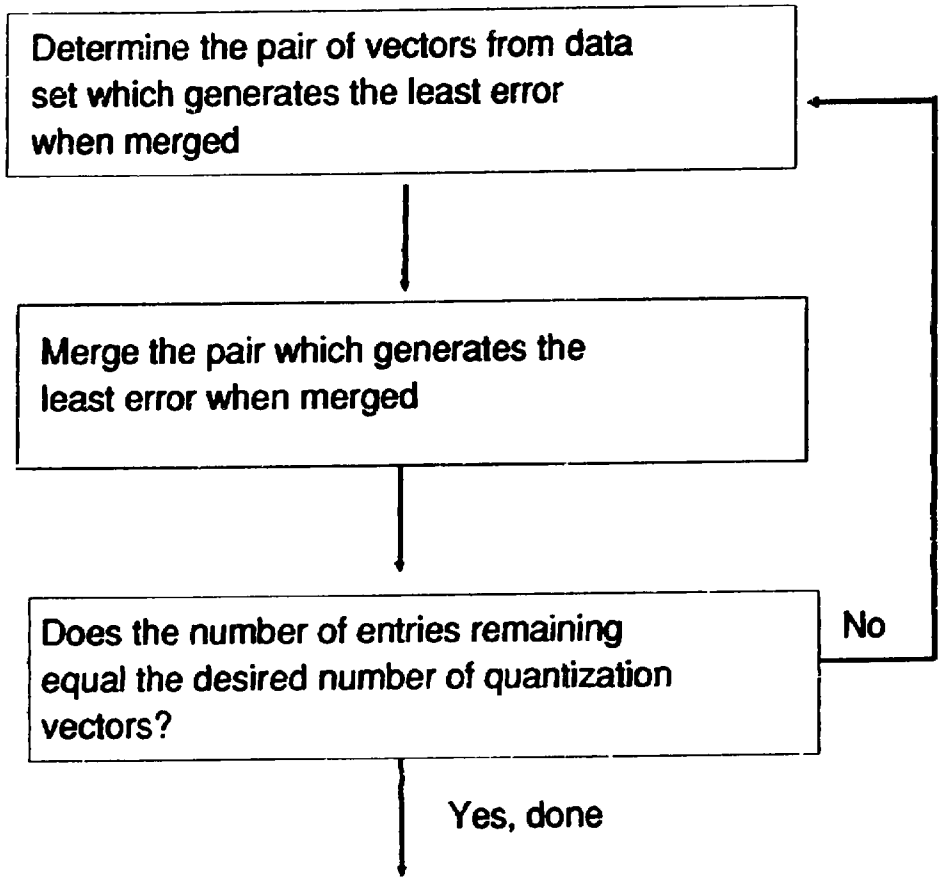


Figure 1. Block Diagram of Full Search PNN

distance calculations would be required on the second iteration, and 998 iterations required on the third iteration. This procedure continues iteratively until the desired number of quantization vectors are reached. Thus, the total number of distance calculations is given by

$$TOT\_DIST\_CALC = N*(N-1)/2 + N-1 + N-2 + \ldots P \quad (2)$$

where TOT_DIST_CALC is the total number of distance calculations required, N is the input data size, and P is the desired number of quantization vectors.

The first term on the right hand side (RHS) of equation (2) dominates the equation and it is obvious that the full search implementation of PNN has computational requirement proportional to O(N**2).

2.2 Fast Implementation of PNN Algorithm

If the requirement to merge the absolute closest pair of vectors at each step is relaxed, as long as vectors get merged eventually then a fast implementation of the PNN algorithm can be derived (Reference 2). This approximated PNN algorithm is called the "fast" PNN (FASTPNN) algorithm. For most applications the error introduced by the FASTPNN approximation of PNN is considered tolerable (reference). The "fast" implementation of PNN was used in this study.

Figure 2 describes the sequence of operations in FASTPNN. The FASTPNN algorithm initially constructs a set of entries from the input data, then iteratively merges pairs of entries until a specified number of quantization vectors are reached. At each iteration, the algorithm proceeds by recursively splitting sets of entries, or buckets, into pairs of buckets until the total number of entries in each bucket is at or below a prespecified number. From each bucket, a candidate pair of entries is nominated that, if merged, provide the smallest increase in quantization error of all pairs within the bucket. A specified percentage of the pairs are selected and merged, with the individual buckets merged into a single bucket to complete an iteration. The sequence is repeated until the total number of entries is equal to the specified number of quantization vectors. The value vectors in the final set of entries define the derived quantization vectors.

The computational savings in FASTPNN is derived in the algorithm's ability to "intelligently" split the global data set among a specified number of localized regions or buckets. Merging of vectors are then performed independently and with more efficiency within these localized regions (buckets).

A key parameter in FASTPNN is BUCKETSIZE, the maximum number of vectors per bucket. BUCKETSIZE plays a key role in determining the number of distance calculations involved in finding the closest vector pair. Since FASTPNN requires that the data be split evenly across all buckets then the total number of buckets required for a particular iteration equals the number of input vectors divided by BUCKETSIZE.

$$BUCKET\_NUM = N/BUCKETSIZE \quad (3)$$

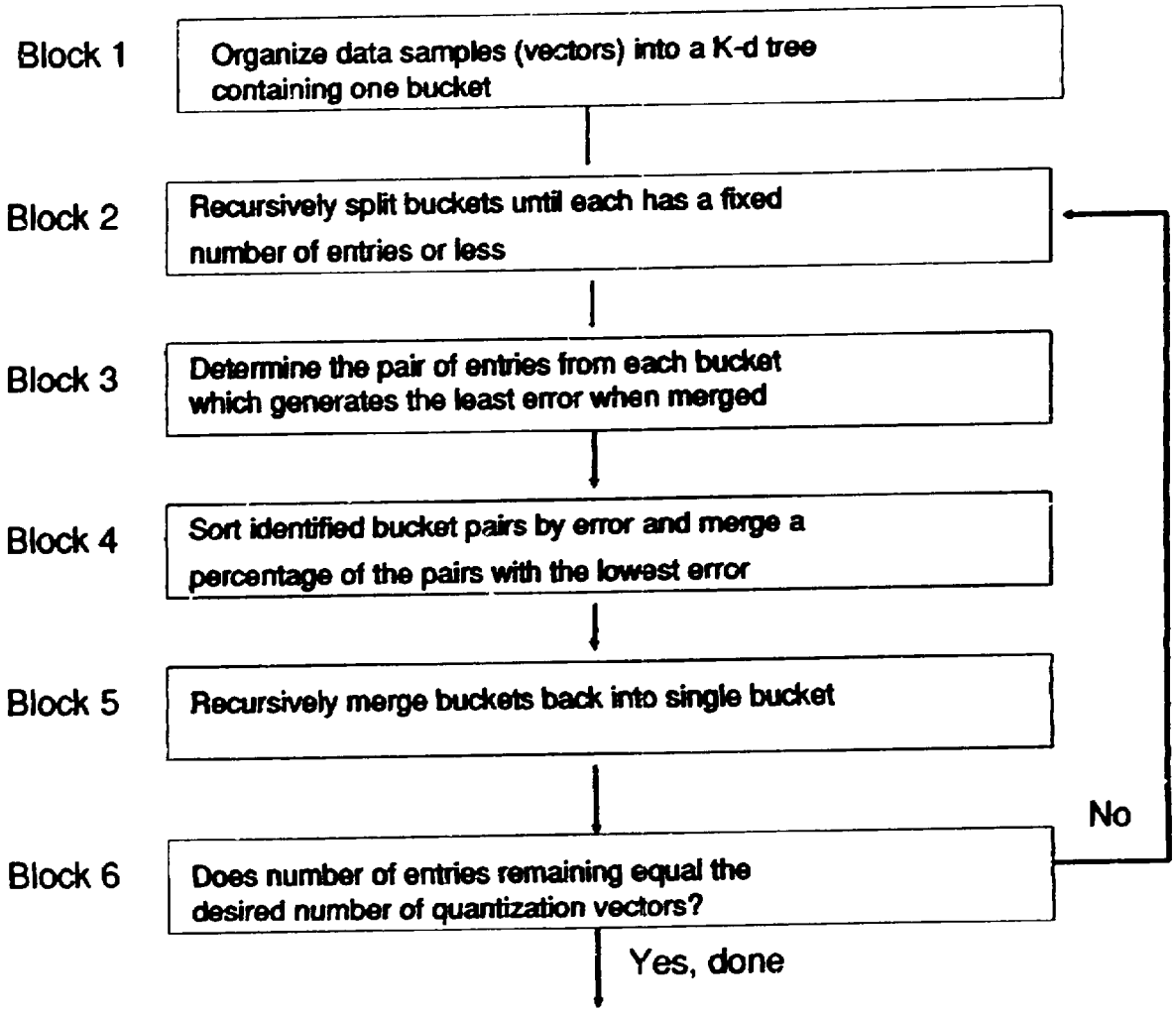


Figure 2. Block Diagram of FASTPNN Algorithm

where BUCKET_NUM equals the total number of buckets for a data set of size N, and bucket containing BUCKETSIZE entries. From equation (1), to find the closest vector pair in each particular bucket requires that (BUCKETSIZE * (BUCKETSIZE - 1)) / 2 weighted distance calculations be performed. Thus, the total number of distance calculations for the complete data set on the first iteration equals the number of distance calculations per bucket times the number of buckets, or

$$DIST\_CALC = (BUCKETSIZE * (BUCKETSIZE - 1)/2) * N/BUCKETSIZE \quad (4)$$

Equation (4) reduces to

$$DIST\_CALC = N * (BUCKETSIZE - 1)/2 \quad (5)$$

where DIST_CALC is the total number of distance calculations required on the first iteration of FASTPNN. Thus, it is readily seen from equation 5 that as BUCKETSIZE decreases the number of calculations goes proportionally down. In the "worst-case" limiting case, if BUCKETSIZE is equal to N (all of the entries are in one bucket) we get the same result as obtained in equation (1) for the full search implementation.

As a simplified example of the magnitude by which FASTPNN reduces the number of distance calculations, let's suppose as we did in the last section that we begin PNN with 1000 entries. If we assume that BUCKETSIZE is ten, then only 4,500 distance calculations are required on the first iteration to find the closest vector pairs. This is significantly lower than the 499,500 distance calculations required by the full search implementation on the first iteration.

The number of vectors merged per iteration is given by the term KDMERGE * BUCKETNUM where KDMERGE is a fixed percentate of the top vector pair candidate from each bucket.

Thus, on the second iteration, the number of distance calculations will be reduced to (N - KDMERGE*N/BUCKETSIZE) * (BUCKETSIZE -1) /2

Although, the number of distance calculations in determining the candidate vector pairs for merging are greatly reduced by using FASTPNN, there are other computational "overhead" factors associated with the FASTPNN algorithm which are not present in the full search implementation of PNN. From Figure 2, these include initialization of the K-d tree data structure and splitting and merging of buckets. Despite these additional "overhead" factors, it has been shown that FASTPNN has computational efficiency proportional to O(N log N).

3. SOFTWARE/BENCHMARK IMPLEMENTATION OF FASTPNN

As stated in the introduction, the FASTPNN algorithm was delivered to the Government coded in the C programming language. Figure 3 shows a subprogram calling tree of the FASTPNN algorithm. Note from Figure 3, that the FASTPNN subprogram structure divides into 3 main parts. These three parts include BuildKDtree, MergedownKDtree, and DestroyKDtree.

BuildKDtree contains the subroutines which involve building and initialization of the K-d tree data structure. The K-d tree is the data structure which permits the partitioning of the global data set into buckets (localized regions) from which nearest neighbor searches can be performed independently. With reference to Figure 2, the functionallity of BuildKDtree is described by block 1 of of the block diagram.

MergeDownKDtree contains the routines which reduce the K-d tree built in Buildkdtree to the specified number of centroids (vectors). With reference to Figure 2, the functionallity of MergeDownKDtree is described by blocks two through six of the block diagram.

DestroyKDtree contains the routines which permit the deallocation of memory which was dynamically allocated in BuildKDtree and MergeDownKDtree. The functionallity of DestroyKDtree is not described in Figure 2, as it is not a necessary part of the FASTPNN algorithm. DestroyKDtree is merely provided to implement the good software practice of deallocating objects no longer in use.

3.1 C Implementation Structure of FASTPNN Benchmark

To permit timing of the C coded FASTPNN algorithm, FASTPNN was integrated with a timer to form a benchmark. The timer measured the process CPU time utilized while running FASTPNN. The basic strategy in timing the FASTPNN algorithm was to have a main (driver) program obtain the initial time just before making a function call to FASTPNN, followed by another call to the timer just prior to executing FASTPNN. The difference between these two times is the time of interest (elapsed time). Because timers use system dependent resources, a separate timer function was written for both the VAX 11/780 and the MIPS MAGNUM 3000. Appendix A provides the C source code listing of the FASTPNN benchmark.

Figure 4 shows the module structure of the C coded FASTPNN benchmark. The FASTPNN benchmark is composed of five modules. Two of the five modules are header files. Header files are used in C to contain definitions and declarations which are to be shared among different files (modules). The arrows in Figure 4 are used to indicate the dependency of particular modules on the header files. For instance, the arrow from FASTPNN.c to FASTPNN.h indicates that FASTPNN.h is to be "included" into FASTPNN.c. Header file Timer.h contains the interface to the system routines which perform the timing of the benchmark. Header file FASTPNN.h contains the data structure declarations which are used in FASTPNN.c and, to a limited extent, in Main.c. The main program, Main.c, is used to read the data, perform the timing of FASTPNN, and output the results.

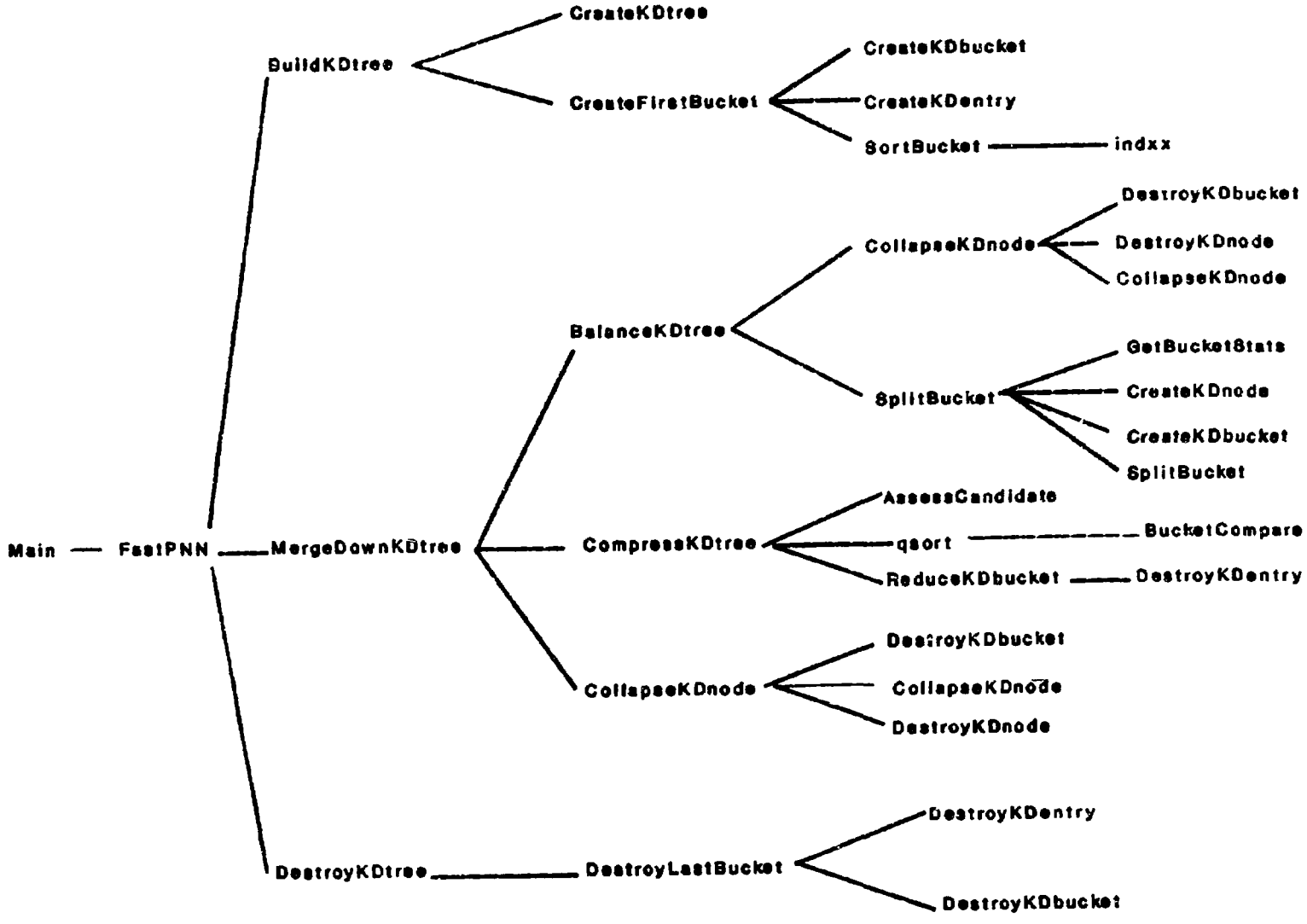


Figure 3. FASTPNN Subprogram Calling Tree

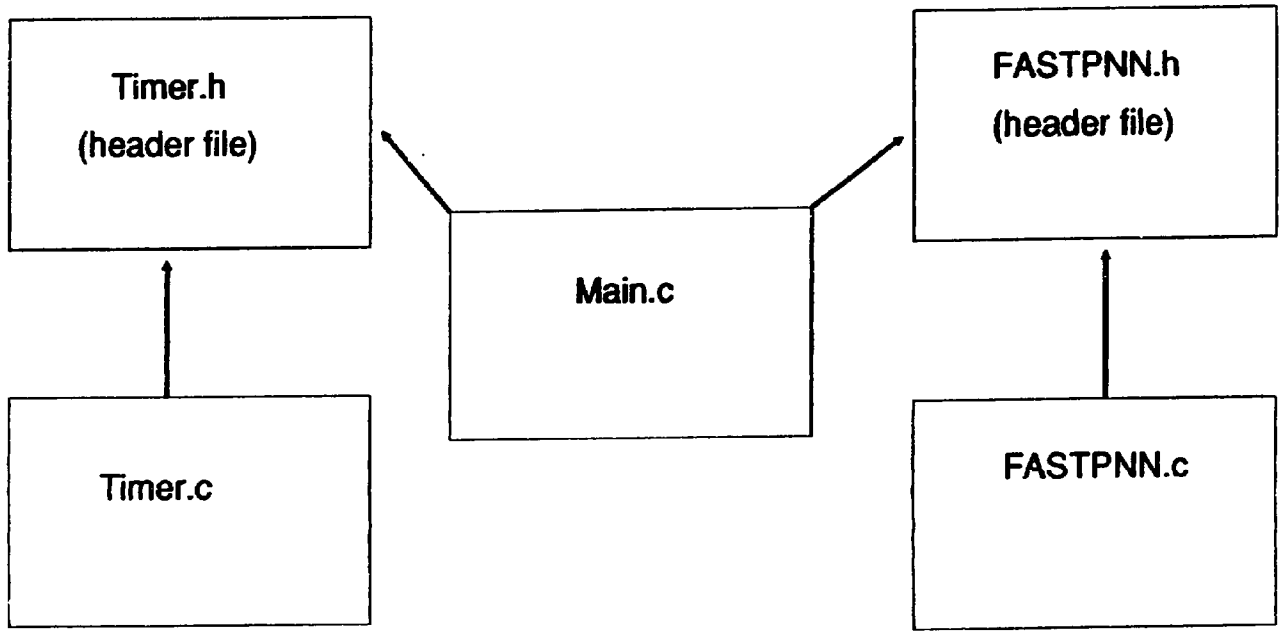


Figure 4. C Module Breakdown of FASTPNN Benchmark

3.2 Ada Implementation of FASTPNN Benchmark

Presented in this section is the organization structure from a module (or compilation unit) level of the C to Ada translated FASTPNN benchmark. The details of the actual coding from C to Ada are addressed in section 4 of this report. Appendix B provides the source code listing of the Ada coded FASTPNN benchmark.

The Ada coded FASTPNN algorithm was integrated with a timer to form a benchmark. The timer measured the process CPU time utilized while running FASTPNN. As with the case of the C benchmark, a separate timer (body) was written for both the VAX 11/780 and the MIPS MAGNUM 3000.

Figure 5 shows the overall module structure of the Ada coded FASTPNN benchmark. Five packages were used in the benchmark implementation. Data_Struct_Pkg contains the declarations of all data types used in the benchmark. Packages Build_Pkg, MergeDown_Pkg, and Destroy_Pkg contain all of the FASTPNN subroutines presented in Figure 3. Package Timer_Pkg contains the sytem dependent routines which permit benchmark timing to be performed. The arrows in Figure 5 are used to indicate the dependency of the particular modules with each other. For instance, the arrows pointing from FASTPNN to Build_Pkg, MergeDown_Pkg, and Destroy_Pkg indicates that these three packages must be "withed into" Build_Pkg. The main program is used to input the data, time FASTPNN, and output the results.

Figure 6 shows the "package location" of each of the FASTPNN subroutines. Build_Pkg contains all of the routines which are nested in the Buildkdtree portion of Figure 3. MergeDown_Pkg contains the routines which are nested in the MergeDownKDtree portion of Figure 3. Destroy_Pkg contains the routines which are nested in the DestroyKdtree portion of Figure 3. Listed in the Ada specification portion of each of the above three packages, are the routines which must be visible outside of their local package usage. Note from Figure 3, that since FASTPNN calls procedures BuildKDtree, MergeDownKDtree, and DestroyKDtree, these three procedures are all included in the specification (as opposed to body) portion of their respective packages. Procedure CreateKDbucket is included in the specification portion of Build_Pkg, since it called by a routine (SplitBucket) outside of Build_Pkg. Procedures DestroyKDnode, DestroyKDbucket, and DestroyKDentry are included in the specification portion of Destroy_Pkg since they are called by routines outside of Destroy_Pkg.

3.3 Benchmark Data Set

TASC provided the Government with an input data set to be used for executing the FASTPNN benchmark. The input data set consisted of a single chip of SAR signature prediction data of a B52 aircraft. A chip is a subregion of interst extracted from a larger image (i.e 1K x 1K). Chips typically range in size from 64x64 to 128x128 pixels. An image plot of the B52 aircraft chip is displayed in Figure 7. This data was obtained by using a prediction tool developed at TASC for predicting SAR returns from various targets.

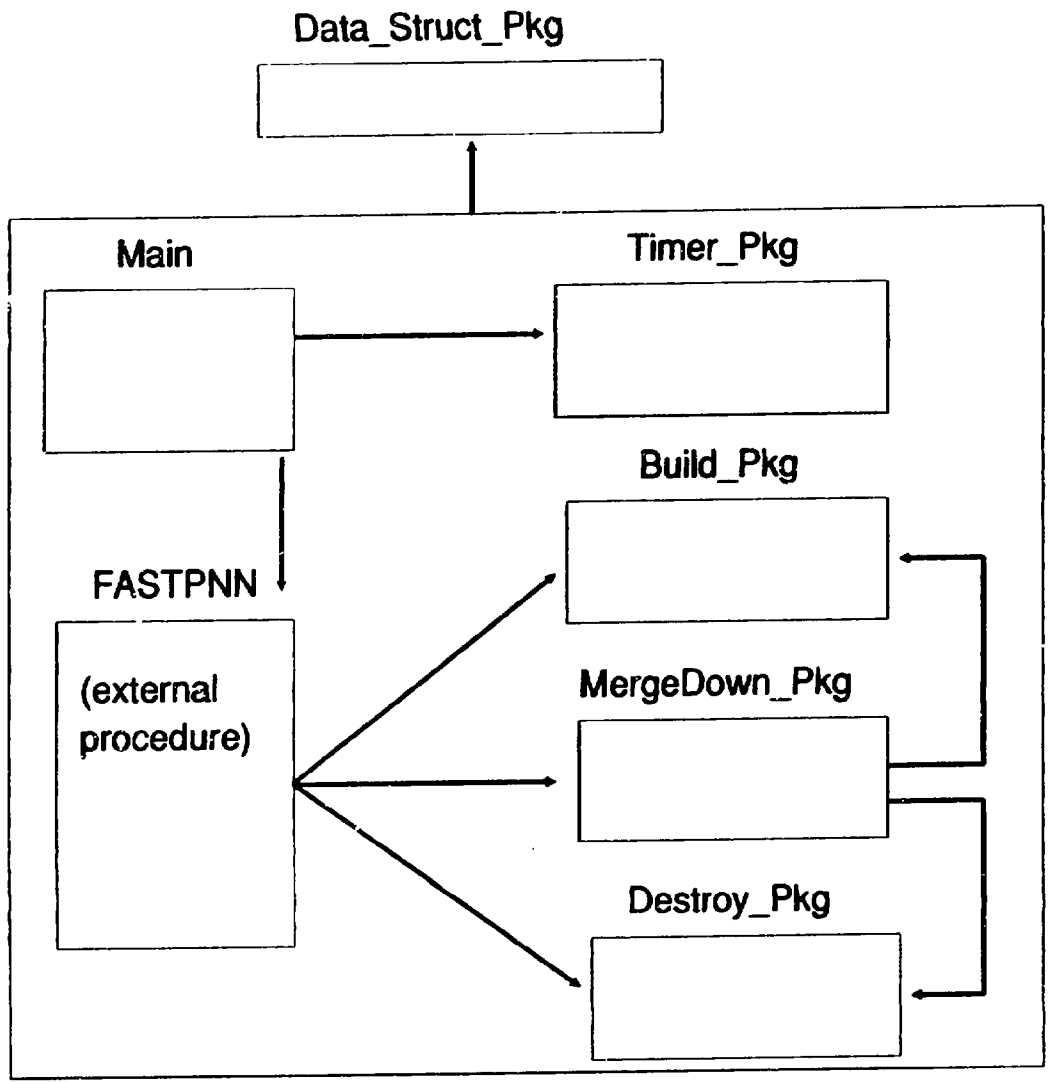


Figure 5. Ada Module Breakdown of FASTPNN Benchmark

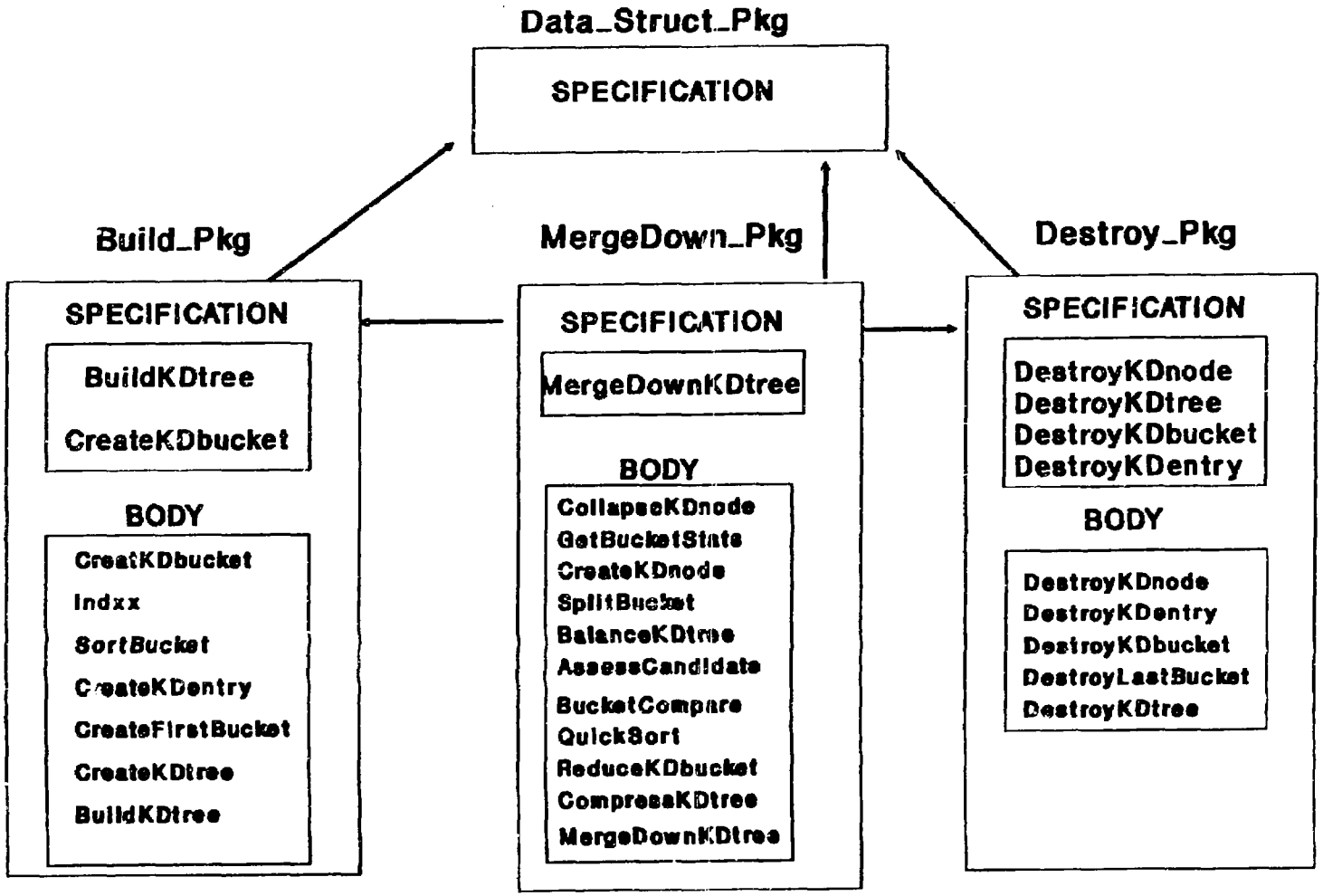


Figure 6. Ada Package Implementation of FASTPNN

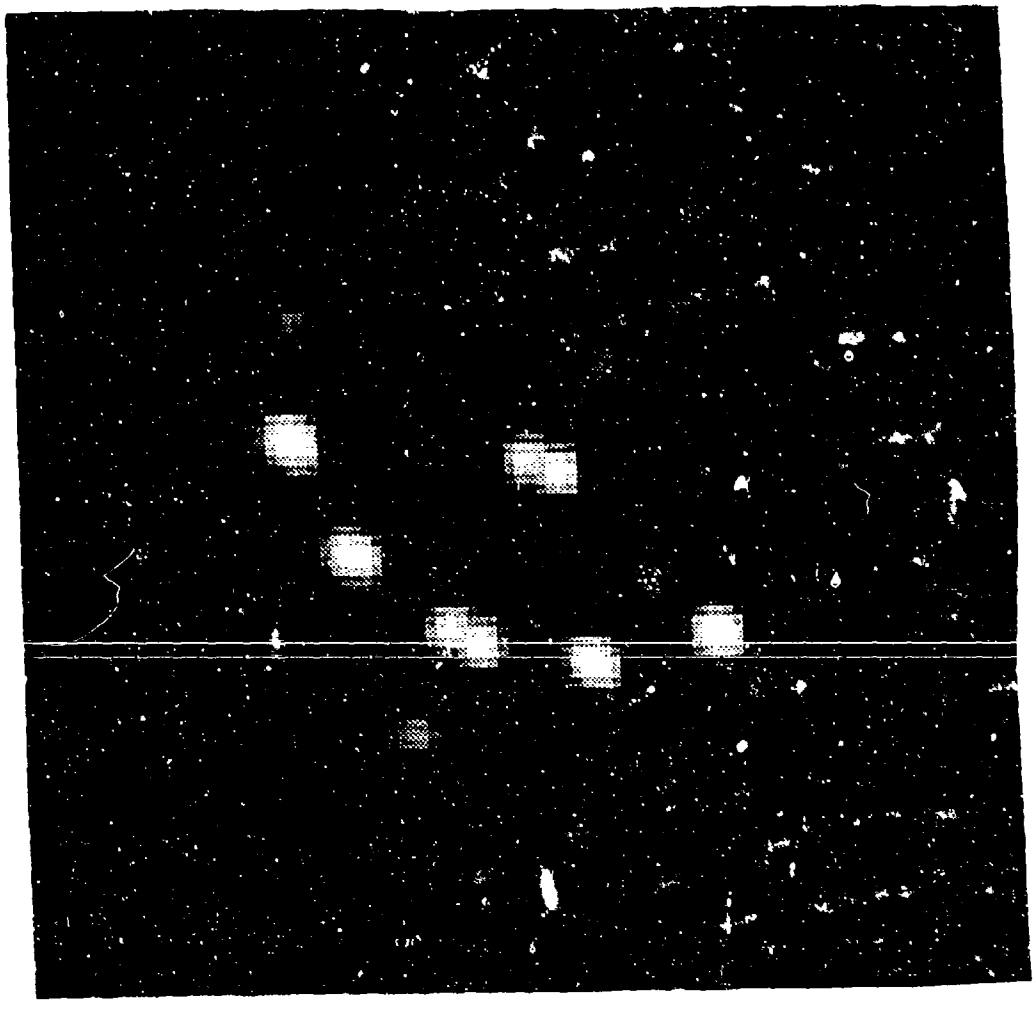


Figure 7. Image Plot of FASTPNN Input Data Set

The input SAR data set (chip) contained 96x96 samples (vectors). Each sample contained 3 components· an x coordinate position, a y coordinate position, and a gray level (weight). Thresholding was used to reduce the intial data set of 5184 vectors to 1,675 vectors. The purpose of thresholding is to reduce the data set by eliminating vectors with non significant weight. The reduced set of 1,675 vectors was the actual input to the FASTPNN algorithm.

4. ADA TO C TRANSLATION STRATEGY.

To establish a uniform framework for recoding the FASTPNN benchmark from C to Ada, a C to Ada translation strategy was adopted. Because the goal is to compare the inherent efficiency of Ada with C, no attempt was made in the translation process to improve the execution efficiency of the Ada source code beyond that of the C source code. Thus, a direct "line by line" translation strategy was adopted.

4.1 Data Structures

The first step in the translation process was to "map" the individual C data structures to "equivalent" Ada data structures. In general, the mapping between C and Ada data stuctures is relatively straightforward. Table 1 below highlights the main data structure mappings used in this study.

TABLE 1. MAPPING OF C TO ADA DATA STRUCTURES

C	Ada
Dynamic Array (*)	Unconstrained Array
Pointer Type	Access Type
Dynamic Array of Pointers (**)	Unconstrained Array of Access Types
Struct	Record
Union	Record with Variant Part

Table 2 provides an example of the data structure mapping techniques presented in Table 1 by presenting two of the translated data structures which were extracted from the FASTPNN algorithm. The left hand side of Table 2 shows the original C coded data structure and the right hand side of Table 2 shows the Ada translated data stucture. These data structures can be found in Fastpnn.h (Appendix A) and Data_Struct_Pkg (Appendix B) of the C and Ada source code respectively.

The sections below provide a brief explanation of the data structure mappings displayed in Table 1 and used as examples in Table 2.

4.1.1 Dynamic Arrays/Unconstrained Arrays

C Implementation. Structure componets *mean, *wmean, and *wsqmn of struct kdentry in Table 2a all correspond to dynamic arrays containing floating point numbers. The number of array components in each of the above arrays is constrained at run time to equal the dimension of the problem's application (i.e. if the problem dimension is two, there would be two components for each array). Thus, in the case of FASTPNN, the use of dynamic arrays permits the flexiblity to use FASTPNN for a generalized n dimensional application. At run-time, memory space is dynamically allocated by making a call to the libray function calloc with the dimension size passed as a parameter to the function calloc.

Ada Implementation. In Ada, one also could also dynamically allocate the arrays *mean, *weight, and *wsqmn by using allocators. However, Ada

TABLE 2. EXAMPLE OF C TO ADA DATA STRUCTURE MAPPING

Table 2a

C

```
struct kdentry {
   struct  kdentry   **next;
   int               splitleft;
   float             weight;
   float             *mean;
   float             *wmean;
   float             *wsqmn;
};
```

ADA

```
type Kdentry;
type kdentry_ptr_type is access kdentry;
type kdentry_ptr_array_type is array (integer range <>)
                                        of Kdentry_ptr_type;

type mean_array_type is array (integer range <>) of float;

type kdentry is
   record
      next_array : Kdentry_ptr_array_type(0..dim-1);
      splitleft : integer;
      weight : float;
      mean_array : mean_array_type(0..dim-1);
      wmean_array : mean_array_type(0..dim-1);
      wsqmn_array : mean_array_type(0..dim-1);
   end record;
```

Table 2b

C

```
struct kdelem{
   int type;
   union {
      struct kdnode   node;
      struct kdbucket bucket;
    } norb;
};
```

ADA

```
type kdelem (data_structure) is
   record
      typee  : integer;
      case data_structure is
          when kdnode_type =>
              node : kdnode;
           when kdbucket_type =>
              bucket : kdbucket;
       end case;
    end record;

norb_n : kdelem(kdnode_type);
norb : kdelem(kdbucket_type);
```

permits an easier and more efficient method to derive most of the flexibility of the dynamic array through use of the unconstrained array. The unconstrained array enables one to treat arrays which have the same characteristics but differ only in their size, as equivalent types. Type mean_array_type in Table 2a is an example of an unconstrained array type.

4.1.2 Pointer Types/Access Types

Pointer types in C map to access types in Ada.

4.1.3 Dynamic Array of Pointers/Unconstrained Array of Access Types

C Implementation. Dynamic arrays of pointers were used to maintain linked list pointers in multiple dimensions. As explained in section 4.1.1, the size of the arrays are dynamically allocated to equal the dimension of the application problem. In Table 2a, structure component **next provides an example of a dynamic array of pointers.

Ada Implementation. Unconstrained arrays of access types were used to maintain linked list pointers in multiple directions. In Table 2a, record componet next_array provides an example of an unconstrained array of access types.

4.1.4 Structures/Records

Structures in C map directly to records in Ada.

4.1.5 Structures Containing Unions/Variant Records.

C Implementation. Unions, like structures, contain members whose individual data types may differ from one another. However the members that compose a union all share the same storage area within the computer's memory, whereas each member within a structure is assigned its own unique storage area. Table 2b provides an example of a structure containing a union.

ADA Implementation. Ada does not contain an equivalent data structure directly mapable to the C union isolated (not contained inside another data structure). However, the Ada variant record maps closely to the case where a where a C union is contained within a C structure. In this case, the "union" portion is represented by the variant part of the record. Table 2b provides an example of a variant record.

4.2 C to Ada Coding Dualities

The final step in the C to Ada coding strategy was to highlight the significant coding dualities between C and Ada. Table 3 below outlines a non exhaustive list of coding dualities between C and Ada.

TABLE 3. Miscellaneous C to Ada Issues

C	Ada
Loop Statements - for - while	Loop Statements - for - while
Break	Exit
Conditional Statements - if - if/else - if/else if/else if/.../else	Conditional Statements - if/end if - if/else/end if - if/elsif/elsif/.../else/end if
- switch	- case
Functions	Procedures or Functions
Pointer Type Operations - *p - p->structure_component - &p	Access Types - p.all - p.record_componet - p'ADDRESS
Dynamic Allocation - calloc, malloc	Dynamic Allocation - new
Deallocation - Cfree	Deallocation - Unchecked_Deallocation
Logical Operators - && - ||	Logical Operators - and then - or else

4.2.1 Loop Statements

Loop statements map very closely between C and Ada and only involve straightforward syntax modifications to convert from one language to the other

4.2.2 Conditional Statements

Conditional statements map very closely between C and Ada and only involve straightforward syntax modifications to convert from one language to the other

4.2.3 "Break"/"Exit" Statement

The "break" statement in C and the "exit" statement in Ada permit the innermost enclosing loop to be exited immediately without "testing" at the bottom or top of the loop.

4.2.4 Subprograms

C Implementation. All subprograms in C are implemented as functions. A C function returns a single value as a result of a call.

Ada Implementation. A subprogram in Ada can be implemented either as a function or a procedure. The following guidelines were used to determine whether a function or a procedure was used. A procedure was used whenever two or more parameters were to be modified through a subroutine call. If one or less parameters was to be modified then either a function or a procedure was used. The decision to choose either a function or a procedure was based upon the characteristics of the subprogram. The following coding convention was used when deciding upon a function or a procedure. If the state of the input argument was to be modified then a procedure call was made. Thus, if the subprogram could be written as an Ada "in-out" variable, then an Ada procedure was used. If just some value was returned, then a function was used. Thus, if the variable that needed to change, was characteristic of an Ada "out" parameter, then an Ada function was used.

4.2.5 Pointer Operations

C Implementation. The unary operator * is called the indirection operator. When applied to a pointer, the indirection operator accesses the object that the pointer points to. For example, if a pointer variable, p, points to an integer of value 2, then the statement x = *p assigns a value of 2 to the integer variable x. If we assume that the pointer variable, p, points to a structure which contains two members (structure components), say member_1 and member_2 respectively, then the statement x = p -> member_1 assigns the value of structure component member_1 to the variable x, and the statement y = p -> member_2 assigns the value of structure component member_2 to variable y. The unary operator &, when applied to a variable, gives the memory address of the object. For example, the statement p = &x assigns the address of x to the pointer variable p.

Ada Implementation. The word "all" is used to access objects pointed to by an access type variable. For example, if an access variable type, p, points to an integer of value 2, then the statement x := p.all assigns a value of 2 to the integer variable x. If we assume that the access variable, p, points to a record which contains two members, say member_1 and member_2 respectively, then the statement x := p.member_1 assigns the value of record component member_1 to the variable x and the statement y := p.member2 assigns the value of record componet member_2 to the variable y. To obtain the memory address of a variable, the attribute 'Address from package system must be used.

4.2.6 Dynamic Allocation

C Implementation. The function calloc was used in conjunction with the function sizeof to obtain blocks of memory dynamically. As an example,

the statement

```
ip = calloc(n, sizeof(int))
```

allocates memory for an array having n elements, with each component having a length equal to number of bytes used by the host machine in representing an integer (usually 2). A pointer ip is returned to these n*2 (assuming an integer is 2 bytes) bytes of uninitialized storage, or NULL if the request cannot be satisfied.

Ada Implementation. Dynamic allocation is performed in conjunction with the "new" statement. Below is the code to translate the same example as presented in the C case above into Ada

```
type integer_array_type is array (1 .. n) of integer
ip := new integer_array_type;
```

The first statement of the above code declares an array of integers of n elements long of type integer_array_type. The second statement uses the keyword "new" to create a designated object of type integer_array_type and it stores the memory address of the newly created object in access type ip.

4.2.7 Memory Deallocation

Deallocation is used to "free" memory which was previously dynamically allocated but is no longer in use. The storage used by that variable can then be reused to allocate a new variable.

C Implementation. The function free deallocates space which was previously dynamically allocated by a call to calloc or malloc. For example, if we assume that memory was previously dynamically allocated by the statement p = (int *) calloc(n, sizeof(int)) hen the statement free(p) frees the space pointed to by p.

ADA Implementation. Unchecked_Deallocation is one of the four predefined generic units that are provided by every implementation of the Ada language . In the Ada language, allocated variables are deallocated by calling an instance of the generic procedure Unchecked_Deallocation. A call on an instance of Unchecked_Deallocation takes one variable, which is a variable of access type. The allocated variable designated by the parameter is deallocated, and the parameter is set to NULL. For example, assume that memory was previously dynamically allocated by the statement p := new Integer_Array_Type; where Integer_Array_Type is an array of integers. To deallocate the array of integers, it is first necessary to instantiate the generic procedure Unchecked_Deallocation by the statement "procedure FREE is new Unchecked_Deallocation (Integer, Integer_Array_Ptr_Type);" and then the statement "FREE(Integer_Array_Ptr)" is used deallocate the array. Note, that there is no requirement to use the name "FREE" as any name (i.e. Deallocate_Cell, Dispose, etc) could be used.

4.2.8 Logical Operators.

C Implementation. Logical operators joined by the logical operators

&& and || are evaluated left-to-right, but only until the overall true/false has been established. In the case of the && operator, if the left-hand expression evaluates to false, the AND THEN operator returns the value false immediately without evaluating the right-hand expression. In the case of the || operator, if the left-hand expression is true, the || operator returns a true value without checking the other operand.

ADA Implementation. The Ada equivalents of && and || respectively are AND THEN and OR ELSE repectively. These are called short-circuit AND and short-circuit OR respectively. They produce the same results as the plain AND and OR, but they force the computer to evaluate the expression in left-to-right order.

5. BENCHMARK DEVELOPMENT SYSTEMS

This section provides a description of the two development systems used in this study to perform the benchmarking.

5.1 MIPS MAGNUM 3000.

5.1.1 MIPS MAGNUM 3000 Development System. Below is a description in outline form of the MIPS Magnum 3000 workstation used in this benchmarking study

- R3000 Central Processing Unit (CPU) running at 25 MHZ
- R3010 Floating-Point Coprocessor (FPC), running at 25 MHZ
- 32 Kbytes of instruction cache (I-cache) and 32 Kbytes of data cache (D-cache), using 20 ns static RAM (SRAM) with fixed 8-word block-refill size
- ASIC Read/Write Buffer with 8-word buffering
- 16 Mbytes of 100 ns DRAM which supports block-mode transfers with peak data rates of 50 Mbytes per second on writes and 100 Mbytes per second on reads

The R3000 CPU provides 32 general purpose 32-bit registers, a 32-bit Program Counter, and two 32-bit registers that hold the results of integer divide and multiply operations. The R3010 FPC is tightly coupled to the R3000 CPU and can execute instructions in parallel with the CPU. The R3010 contains sixteen, 64-bit registers that can be used to hold single-precision or double-precision values. The MIPS R3000 has 74 instructions while the MIPS R3010 has 20 instructions. However, since eight of the R3000 instructions are common to R3010 instructions, there are a combined total of 86 MIPS instructions. The R3000 can access memory only through simple load/store operations. All MIPS instructions are 32 bits long. Although there is only a single addressing mode (base register plus 16-bit signed displacement), there are numerous individual load and store instructions that can load or store integer data in sizes 8, 16, and 32 bits with signed and unsigned extension.

5.1.2 MIPS Compilers. MIPS compilers support six programming languages including C and Ada. The compiler system has a separate front-end to translate each language and a common back-end to generate optimized machine code. Run-time libraries provide language-dependent functions for each language. The front-ends translate the semantics of each language into an intermediate representation, called U-code. U-code is used by several of the common back-end components.

MIPS uses a common global optimizer, called uopt, for all of their compilers. MIPS categorizes there compiler optimizations into four different levels. Below is a summary of the different optimization levels:

Level 1 - includes peephole and local optimizations

Level 2 - includes level one optimizations, plus the following global optimizations: loop-invariant code motion, strength reduction, common subexpression elimination, and register allocation

Level 3 - includes level two optimizations plus interprocedure register allocation

Level 4 - includes level 3 optimizations, plus procedure inlining

5.1.2.1 MIPS Ada Compiler.

The MIPS Ada compiler provides two global optimizers, namely uopt (discussed in section 5.1.2) and OPTIM3.

OPTIM3 is a high level global optimizer that performs many classical code optimizations and several that are specific to Ada. These include redundant range check elimination and range propagation for elimination of constraint checking.

Version 3.0 of the MIPS Ada compiler was used in this benchmarking study. MIPS Ada 3.0 is intended to support the level 2 optimizations presented in section 5.1.2. However, although the Ada code compiled properly using level 2 optimizations, the run-time execution terminated due to a "segmentation error". Thus, it was necessary to only use level 1 optimizations in compiling the Ada source code.

5.1.2.2 MIPS C Compiler.

The MIPS C compiler used was the 2.11 release. The 2.11 release supports all of the level 4 optimizations presented in section 5.1.2. In this study, the MIPS C code was compiled and executed using level 4 optimizations.

5.2 VAX 11/780.

5.2.1 VAX Hardware.

Below is a description in outline form which summarizes the main features of the VAX 11/780 computer.

- contains a VAX 11/780 processor
- Floating Point Accelerator is optional (used in this study)
- contains an 8 Kbyte cache which hold both data and instructions
- contains as addresss translation buffer (cache) which can hold up to 128 virtual-to-physical page-address translations
- contains an 8 byte instruction buffer

The VAX is the classical example of a Complex Instruction Set Computer (CISC) architecture. There are over 200 different instructions and 7 basic addressing modes. The instruction set operates on integer, floating-point, character-strings and packed-decimal strings, and bit fields. The processor provides 64-bit, 32-bit, 16-bit, and 8-bit arithmetic; instruction prefetch; and an address translation buffer. The CPU includes 16 32-bit general purpose registers for data manipulatuion and the Processor Status Longword for controlling the execution states of the CPU. The VAX used in this study contained an optional high performance floating point accelerator (FPA). The FPA is an independent processor that executes in parallel with the base CPU. The FPA takes advantage of the CPU's instruction buffer to access main memory. Once the CPU has the required data, the FPA overrides the normal execution flow of the standard floating-point microcode and forces use of its own code. While the FPA is executing the CPU can be performing other operations in parallel.

5.2.2. VAX COMPILERS. This section describes both the VAX Ada compile and also the VAX C compiler.

5.2.2.1 VAX ADA COMPILER.

Version 1.5 of the VAX ada compiler was used. The compiler was run with full optimization with regard to time. This optimization includes both local and global optimizations similar to those which are performed by the MIPS compilers discussed in section 5.1.2.

5.2.2.2 VAX C Compiler

The VAX C compiler compiler can perform global and local optimization by, for example, doing global flow analysis, assigning automatic variables to register temporaries, and removing invariant computations from loop, to mention a few. The compiler also does peephole optimizations on the generated machine code.

Version 2.3 of the VAX C compiler was used in this study and run with full optimization with regard to time (as opposed to space).

6. BENCHMARK RESULTS/DISCUSSION.

This section presents the benchmark results obtained by running both the Ada and C coded FASTPNN benchmark discussed in section 3 using the coding rules discussed in section 4 on the development systems discussed in section 5.

It is important to reiterate from sections 5.1.2.1 and 5.1.2.2, that the MIPS C code was compiled and executed using level 4 compiler optimizations while the MIPS Ada code was compiled with level 1 optimizations. Recall, that this is due to the fact that MIPS level 3 and 4 compiler optimizations were not supported in the Ada compiler used (Version 3.0). Further, when the Ada code was executed using level 2 optimizations a fatal segmentation error resulted. Thus, only level 1 optimizations could be used for the Ada.

In an attempt to circumvent the discrepancies between the MIPS C compiler optimizations and the MIPS Ada optimizations, in section 6.2, projections of Ada execution time are made based upon level 4 compiler assumptions. The motivation for such a discussion is that eventually level 4 optimizations will be incorporated in Ada. The level 4 Ada execution time projections are based upon using the scaling factor associated with the C execution improvement when going from level 1 to level 4 optimizations. Thus, the level 4 Ada execution time projections are obtained by multiplying the Ada level 1 execution time by the C level 4 execution time, and then dividing this result by the C level 1 execution time.

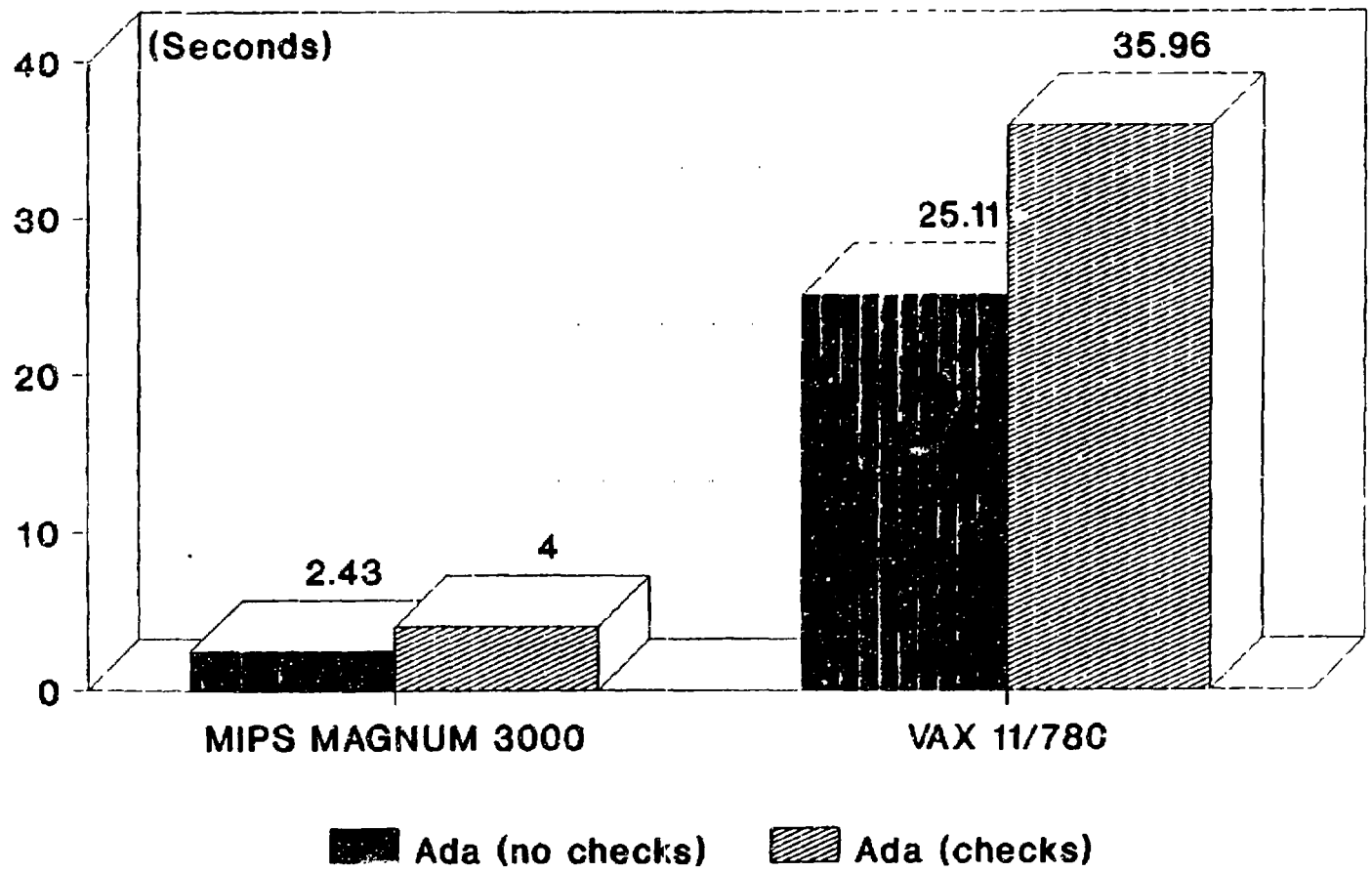
6.1 Ada Without Checks Versus Ada With Checks

To support the use of exceptions, Ada performs run time checks to determine whether an exception should be raised. In practice, a clever compiler can determine that many of the checks can be safely omitted. Nevertheless, a compiler may continue to generate checks that a programmer knows are unnecessary, and these checks may make a critical difference in the execution time of a program. The different run time checks include access checks, discriminant checks, index checks, length checks, range checks, division checks, and overflow checks.

To investigate the execution time overhead in performing checks, the FASTPNN Ada code was executed for both the case where Ada checks were performed and also the case where all Ada checks were suppressed (no checks).

Figure 8 contrasts the execution efficiency of the Ada code run for the case where Ada run-time checking was performed versus the case where run-time checking was suppressed. Figure 8 indicates that Ada run-time checks imposes a significant penalty on the execution efficiency of the Ada for both the VAX and the MIPS. On the VAX, the PNN execution time goes from approximately 25 seconds with checks off to approximately 36 seconds with checks on. This corresponds to a 43% increase in time for performing checks on the VAX. On the MIPS, the execution time goes from approximately 2.4 seconds with checks suppressed to 4 seconds with checks preformed. This corresponds to a 65% increase in time for performing checks on the MIPS. Thus, the relative penalty for performing Ada run-time checks is moderately higher for the MIPS than for the VAX.

FIGURE 8. EFFECT OF ADA RUN TIME CHECKS ON EXECUTION EFFICIENCY



6.2 C Versus Ada

Figure 9 contrasts the execution efficiency of the C code with the Ada code. The relative efficiency of C versus Ada depends on which machine the code is executed on. If the code is executed on the VAX, than the Ada executes more efficiently than the C. For the case of the VAX, the Ada executes approximately 39.5% quicker than the C when Ada checks are suppressed and 13.3% quicker than the C when Ada checks are performed. On the other hand, if the code is executed on the MIPS the C code runs quicker than the Ada code. For the MIPS, the C code runs approximately 40% quicker than the Ada code when Ada checks are suppressed and 114% quicker than the Ada code when Ada checks are incorporated.

Figure 10 contrasts the execution efficiency of Ada versus C for the case where level 4 Ada compiler optimizations are projected using the scaling technique discussed in section 6. Note from Figure 10 that the Ada execution time for the case where Ada checks are not incorporated is nearly identical to the C execution time.

Based on Figures 9 and 10 it is concluded that there is little or no difference between the inherit execution efficiency of Ada (without checks) and that of C. The actual execution efficiency of Ada versus C is driven by the maturity of the compilers used in the comparison. With regard to the MIPS, the fact that the C code runs quicker than the Ada is attributed to the fact that the present MIPS Ada compiler is not as mature as the MIPS C compiler. This is because the MIPS Ada compiler was run with level 1 compiler optimizations while the MIPS C compiler was run using level 4 compiler optimizations. Figure 10 indicates that eventually when the MIPS Ada compiler matures to the level of the C compiler, the Ada code will execute in nearly identical efficiency (time) as the C code.

The fact that the Ada code on the VAX executes more efficiently than the C code on the VAX is attributed to the hypothesis (this was not proven in this study) that the VAX is less efficient than the MIPS in performing dynamic allocation. Thus, in the case of Ada, where the unconstrained array was used in replacement to performing dynamic array allocation using access types, the resulting Ada code was more efficient than the C code where dynamic allocation had to be performed. On the MIPS, however, where dynamic allocation is performed more efficiently this difference between Ada and C was nullified.

6.3 MIPS Versus VAX

By far the most common method used to gauge the performance of a particular machine is to measure its performance relative to the VAX 11/780. A ratio, expressed in terms of VAX MIPS, is obtained by dividing the time required to execute a given benchmark on the VAX versus the time to execute that same benchmark on a different computer. This ratio is used to express the machine's performance relative to the VAX; the higher the ratio, the better the machine's performance. Figure 11 contrasts the execution efficiency of the MIPS MAGNUM 3000 with the VAX 11/780. Note from Figure 11 that the ratio of MIPS execution time versus VAX execution time is highly dependent on whether Ada or C is used. If C is the language, then the ratio is approximately 22.2 VAX MIPS. If Ada is the language the MIPS ratio

Figure 9. FASTPNN EXECUTION TIME COMPARISON

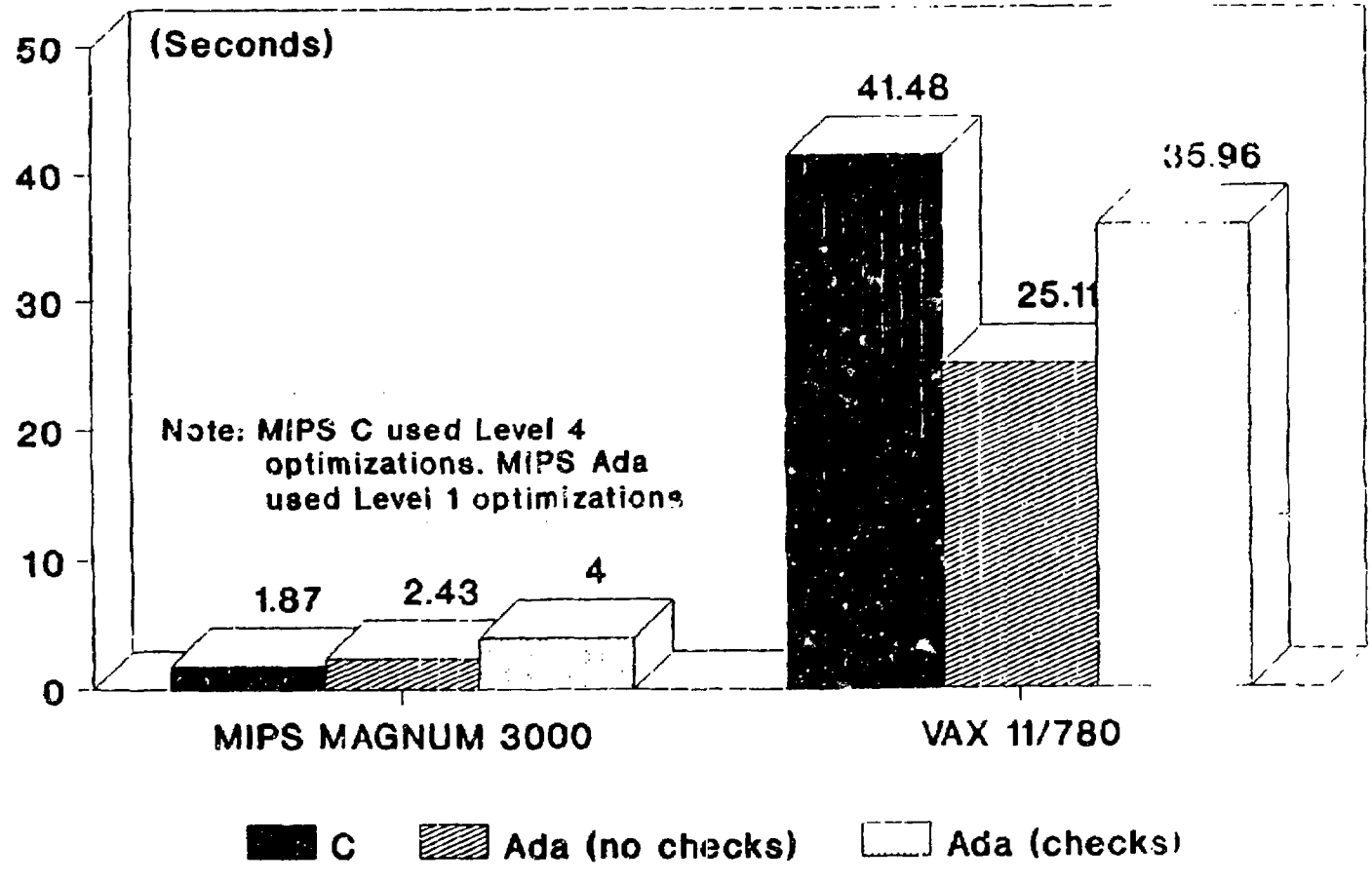


FIGURE 10. FASTPNN EXECUTION TIME COMPARISON BASED UPON ADA LEVEL 4 OPTIMIZATION PROJECTIONS

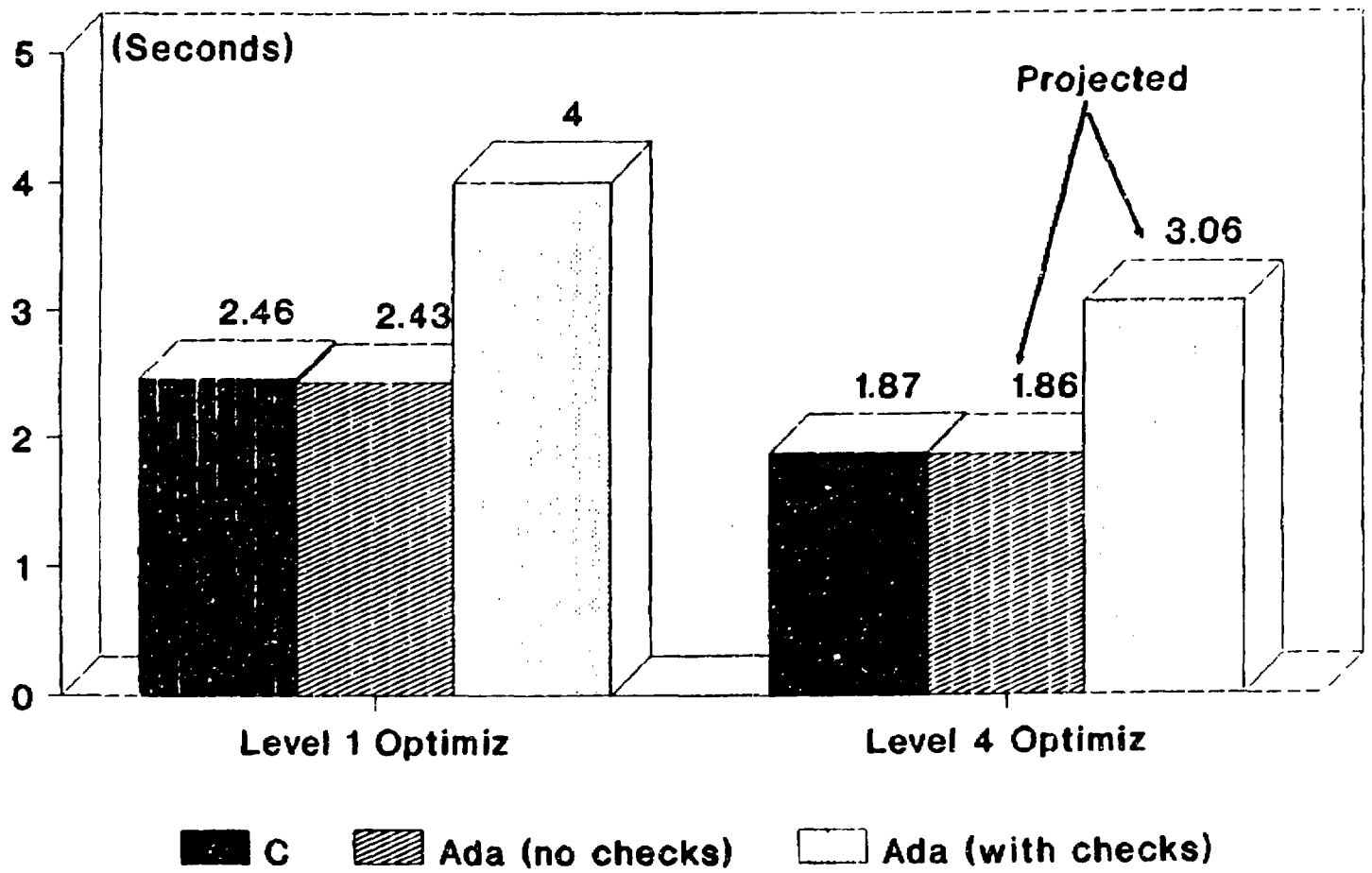
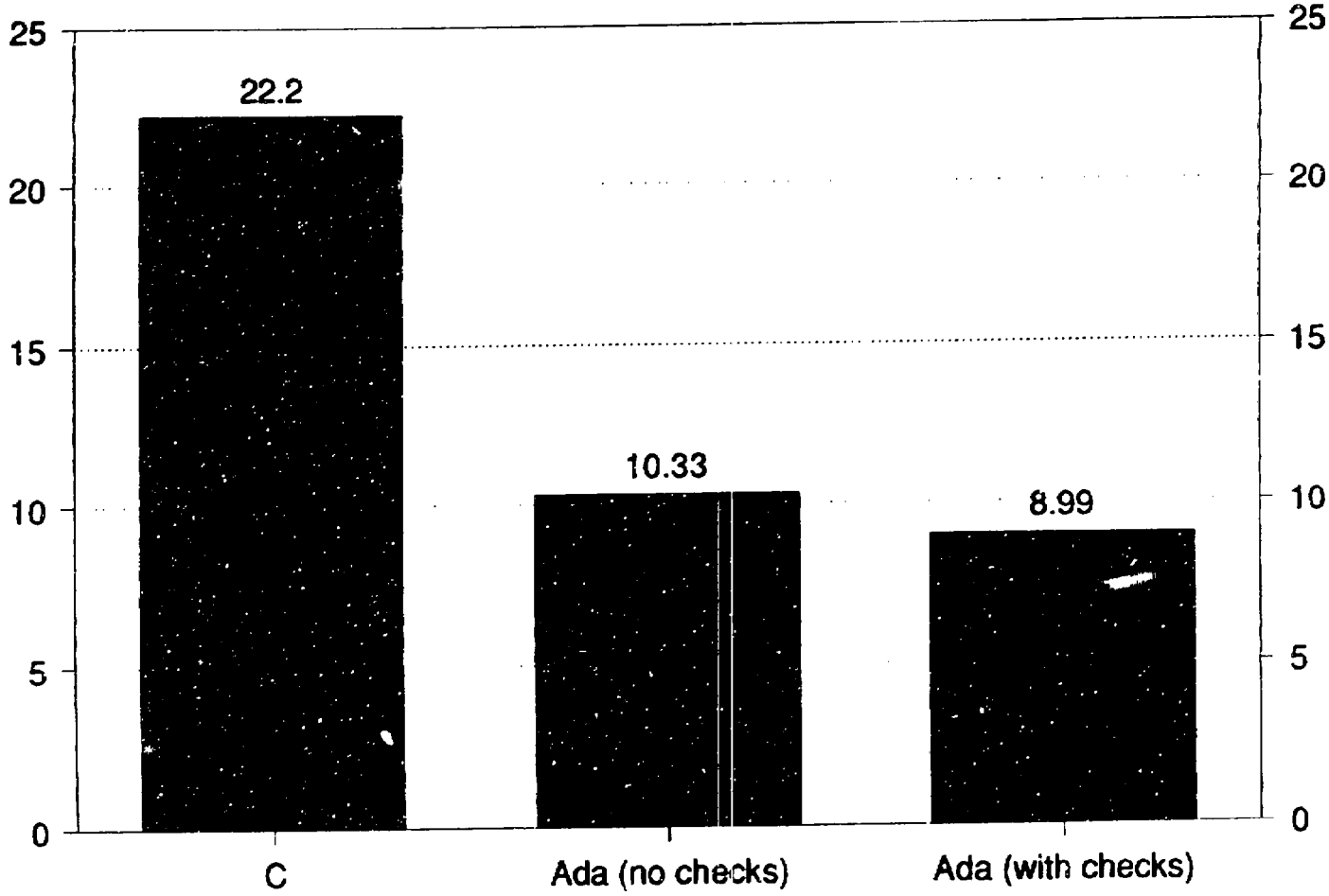


FIGURE 11. RELATIVE THROUGHPUT OF MIPS MAGNUM 3000
(NORMALIZED TO VAX MIPS)



drops off dramatically and is dependent, to a small degree, on whether Ada is run with run-time checks or without run-time checks. If Ada is run without checks than the ratio is approximately 10.3 VAX MIPS. In the case where Ada is run using run-time checks, the ratio reduces to approximately 9.0 VAX MIPS.

The above results indicate that the MIPS MAGNUM 3000 is somewhere between 9 to 22.2 times faster than the VAX 11/780, with the variance in ratio attributed to the compiler. Since C compilers are more mature than Ada compilers, it is concluded that the C run time results are a better indicator of inherit machine performance than the Ada run time results. Thus, it is concluded that the actual speedup of the MIPS over the VAX is reflected most accurately by the C VAX ratio of 22.2 VAX MIPS.

6.4 MIPS Profiler Results

The purpose in profiling is to help identify the areas of code where most of the execution time is spent. In the typical program, execution time is disproportionally spent in relatively few sections of code. Having identified these critical sections of code, it is profitable to improve coding efficiency in those sections.

The results presented in this section were obtained by using the MIPS UNIX profiler. In both the C and Ada case, the profiler output statistics assume 100% cache hits. For each subroutine in PNN, the profiler outputs the following statistics: the total number of cycles used by that routine (CYCLES), the percentage of cycles the routine uses with respect to the total number of program cycles (%CYCLES), the cumulative percentage of cycles (CUM%), the total number of times each routine is called, and the number of cycles used by the routine per call (CYCLES/CALL). Note that there is a direct relation between cycles used and execution time; simply divide the cycles used by the clock freqency (25 MHz) to get the actual time.

Since the profiler results presented in this section are categorized by subroutine, a brief description of the seven most time consuming subroutines contained in FASTPNN are presented in Table 4. Table 4 can be used to gain insight into the time breakdown of the FASTPNN routines in terms of computational functionallity.

6.4.1 C Profiler Results

In this section the profiler was run on the FASTPNN C code using level 2 compiler optimizations. Table 5 summarizes the profiler results for the seven most time consuming subroutines in FASTPNN. Table 5 lists the seven routines in descending order corresponding to their overall contribution with regard to cycles used in the execution of FASTPNN. Note from Table 5 that the seven routines account for approximately 89% of the total cycles used in the entire FASTPNN algorithm. The routine which accounts for the greatest percentage of cycles (time) is GetBucketStats. From Table 4 one can see that GetBucketStats is concentrated on basic floating point mathematics in performing mean and variance calculations. The first four of the routines listed in Table 5 account for over 80% of the total cycles used. Also note from Table 5 that routine Indxx is by far the most time-

TABLE 4. DESCRIPTION OF MOST SIGNIFICANT PNN SUBROUTINES

SUBROUTINE NAME	SUBROUTINE DESCRIPTION	FUNCTIONS TESTED
AssesssCandidate	Determine distortion among all pairs of bucket entries	floating point arithmetic conditional branching linked list traversal
CollapseKDnode	recursive routine used to collapse entries in a Kd tree into a single bucket	linked list manipulation conditional branching recursive calling
GetBucketStats	mean and variance calculation for each bucket	floating point arithmetic array manipulation linked list traversal
Indxx	heapsort on vector position coordinates	array indexing conditional branching
Qsort	sort candidate pairs of entries from each bucket for merging	recursive calling conditional branching
ReduceKDbucket	merge a pair of entries into a single entry	floating point arithmetic linked list manipulation
SplitBucket	recursive routine used to split a K-d tree bucket into two buckets having half as many entries	linked list manipulation recursive calling conditional branching

TABLE 5. MIPS C PROFILER RESULTS

SUBROUTINE	CYCLES	%CYCLES	CUM%	CALLS	CYCLES/CALL
GetBucketStats	11,151,089	25.76	25.76	3277	3403
SplitBucket	9,495,506	21.94	47.70	6617	1435
CollapseKDnode	8,483,026	19.60	67.30	3277	2589
AssessCandidate	5,735,734	13.25	80.55	3343	3343
Indxx	1,944,679	4.49	85.04	2	972,340
qsort	962,354	2.22	87.26	4498	214
ReduceKDbucket	935,071	2.16	89.42	1671	560

consuming routine with regard to the number of cycles required on a per call basis, requiring over 970,000 cycles per call. But because Indxx is only called twice, it accounts for just approximately 4.5% of the total cycles used.

6.4.2 Ada Profiler Results

In this section the profiler was run on the Ada code using level 1 compiler optimizations and suppressing all Ada run time checks. Table 6 summarizes the profiler results for the seven most significant subroutines in FASTPNN. Table 6 lists the seven routines in descending order corresponding to their overall contribution with regard to cycles used in the execution of FASTPNN. Note from Table 6 that the seven routines account for approximately 94% of the total cycles used in the entire PNN algorithm. The routine which accounts for the greatest percentage of cycles (time) is SplitBucket. From Table 4 one can see that SplitBucket is computationally intensive in the areas of recursive calling, linked list manipulation, and conditional branching. The first four of the routines listed in Table 6 accounts for over 84% of the total cycles used in FASTPNN. Also note from Table 6 that routine Indxx is by far the most time consuming routine on a per call basis requiring approximately 1.3 million cycles per call.

6.4.3 Ada Versus C Profiler Results

Table 7 contrasts the performance of the Ada profiler results with the C profiler results for each of the main routines listed in the previous two sections. The seven routines are presented in alphabetical order in Table 7. Table 7 displays the amount of cycles used by each routine, the relative percentage of cycles that the routine contributes to the entire FASTPNN cycle count, and a ranking of each routine corresponding to its relative contribution of cycles used. With the exception of routine Assesscandidate, the results displayed in Table 7 are consistent from the standpoint that the Ada routines use more cycles than the C routines. In the case of AssessCandidate, the C requires more cycles than the Ada. With the exception of routines SplitBucket and GetBucketStats, the relative ranking of the individual routines for C versus Ada is also consistent. In the case of routine SplitBucket, SplitBucket is the most time consuming routine (rank 1) in the case of Ada while, in the case of C, SplitBucket is the second most time consuming routine (rank 2). In the case of routine GetBucketStats, GetBucketStats is the second most time consuming routine (rank 2) in the case of Ada, while GetBucketStats is the most time consuming routine (rank 1) in the case of C.

The last row of Table 7 shows the total cycles from the seven routines combined and the cumulative percentage of cycles this sum comprises of the overall cycles used in the FASTPNN execution. Note, that the total cycles used by the Ada exeeds the total cycles used by the C code. This result is consistent with the execution results previously displayed in Figure 9.

TABLE 6. MIPS ADA PROFILER RESULTS

SUBROUTINE	CYCLES	%CYCLES	CUM%	CALLS	CYCLES/CALL
SplitBucket	15,143,120	29.46	29.46	6628	2285
GetBucketStats	14,259,413	27.75	57.21	3281	4347
CollapseKDnode	8,936,579	17.38	74.59	3281	2724
AssessCandidate	4,977,050	9.68	84.27	3347	1488
Indxx	2,540,485	4.94	89.21	2	1,270,243
QuickSort	1,518,782	2.95	92.16	4498	338
ReduceKDbucket	1,164,919	2.27	94.43	1671	697

TABLE 7. COMPARISON OF MIPS PROFILER RESULTS
(ADA VERSUS C)

	ADA			C		
SUBROUTINE	CYCLES	%CYCLES	RANK	CYCLES	%CYCLES	RANK
AssessCandidate	4,977,050	9.68	4	5,735,734	13.25	4
CollapseKDnode	8,936,579	17.38	3	8,483,026	19.60	3
GetBucketStats	14,259,413	27.75	2	11,151,089	25.76	1
Indxx	2,540,485	4.94	5	1,944,679	4.49	5
QuickSort	1,518,782	2.95	6	962,354	2.22	6
ReduceKDbucket	1,164,919	2.27	7	935,071	2.16	7
SplitBucket	15,143,120	29.46	1	9,495,506	21.94	2
TOTAL	48,540,348	94.43		38,707,459	89.42	

7.0 PROJECTED FASTPNN REAL-TIME REQUIREMENT

The goal of this section is to motivate a FASTPNN real time requirement. The basic strategy used in motivating the FASTPNN real time requirement derived in this section is outlined below:

A. First, determine the MBV real time requirement/goal (how many targets need to be recognized in how many seconds?)

B. Next, determine what percentage of time an MBV system will be spend on FASTPNN

C. Last, multiply the percentage of time spent on FASTPNN (B above) by the overall MBV real-time requirement (A above). FASTPNN must then be able to complete its execution in this derived interval of time

There is currently a program sponsered by the Air Force called Automatic Radar Air-to-Ground Acquisition Program (ARAGTAP) which is focused on establishing a real-time MBV capability. The ARAGTAP goal is to identify 20 to 40 objects (targets) from a high resoltion SAR image in approximately 7 seconds. If we conservatively assume a 20 to 30% alarm rate, this requirement translates to identifying approximately 50 chips (of which 20 to 40 may be actual targets) that range in size from 64 x 64 pixels to 128 x 128 pixels in 7 seconds.

Step B in our strategy to determine a FASTPNN real-time requirement is to determine the relative percentage of the time that the MBV algorithm will spend on FASTPNN. A heuristic reasoning process was used to estimate that FASTPNN should account for approximately 1.86% of the total MBV processing time. The heuristic reasoning process included the following assumptions:

- 10% of MBV should be spent on prescreening/detection and the other 90% should be spent on recognition

- of the remaining 90% of the time spent on recognition, 25% of the processing time should be spent on information extraction algorithms and 75% should be spent on classification/matching algorithms

- of the 25% of the time spent on information extraction, 8.25% of this time should be spent on the FASTPNN algorithm.

Mathematically combining all of the above assumptions (by multiplying), it is found that the FASTPNN algorithm accounts for approximately 1.86% of the total MBV execution time.

Since we are assuming that 50 chips must be processed in 7 seconds, it is determined that the FASTPNN algorithm must be able to process a single chip (as was done in this study) in .0026 seconds.

In section 6 the best case FASTPNN execution time was 1.87 seconds; this is nearly 3 orders of magnitude slower than the FASTPNN real-time requirement previously derived in this section.

8.0 FASTPNN/PNN IMPLEMENTATION CONSIDERATIONS

Based upon the discussion in section 7, it is evident that FASTPNN execution efficiency must be greatly increased if it is to function within real-time MBV constraints.

In section 2.1 it was shown that FASTPNN had compuational requirements that are O(NLOGN) while the full search PNN had computational requirements O(N**2). The fact that the full search algorithm is O(N**2) and that FASTPNN is O(NLOGN) does not necessarily imply that FASTPNN will execute more quickly than full search PNN over all input data sets. In fact, for up to relatively large data set sizes, N, it is highly conceivable that the full search implementation PNN will execute more efficiently than FASTPNN. This is attributed to the fact that there are initializations and other overhead associated with the FASTPNN algorithm. However, as N gets very large, the volume of calculations associated with the full search implementation will dominate all overhead associated with FASTPNN, and the full search implemetation will run slower than FASTPNN. A logical question to ask is: "for a given data set size which implementation of PNN should one use"? For the sake of convenience we will designate the parameter BREAK_EVEN_N to refer to the data set size, N, where the execution time of FASTPNN and full search PNN would be equal. It will be understood that for N < BREAK_EVEN_N, the full search implementation will run quicker than FASTPNN, and for N greater than BREAK_EVEN_N, the full search implementation will run slower than FASTPNN.

It is important to emphasize that BREAK_EVEN_N does not indicate the data set size where both implementations of PNN are equally as good. Given equal execution time for both implementations of PNN, the full search implementation of PNN is superior to FASTPNN since the resulting quantization vectors are optimum. In fact, it is logical to assume that significant computational savings must be obtained to warrant the use of FASTPNN over the full search implementation.

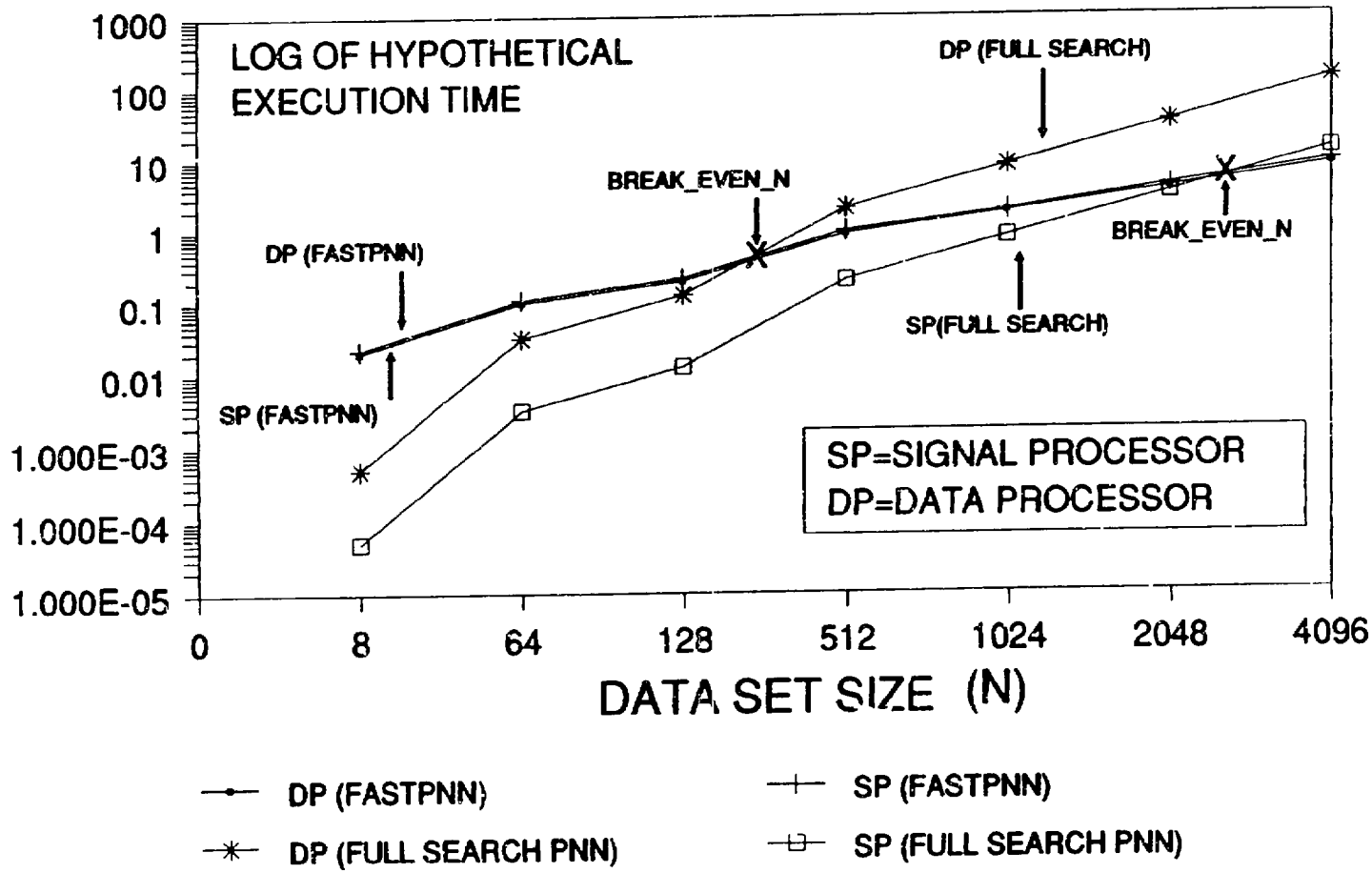
One strong disadvantage of the full search PNN is the large memory requirement which would be required if a large input data set size is used. For example, it was shown by example in section 2.1 that if the input data set contains 1000 entries, then 499,500 distance calculations would be required to be computed in the first iteration. If each distance calculated were to be stored in one word, then this requirement translates to nearly two megabytes of memory.

Intuitively, the parameter BREAK_EVEN_N is dependent on the actual architecture used in executing the PNN algorithm. In this study, where benchmarking was performed on general purpose data processors it was assumed (but not demonstrated) that, for the given input data set, FASTPNN was more time efficent than full search PNN. However, if one were to consider executing the PNN algorithm on a signal processor (as opposed to a data processor), the choice between choosing the full search implementation of PNN versus FASTPNN could certainly be altered. A signal processor will be defined, in the context of this section, as a processor specifically designed to perform sequences of computations/operations unaffected by actual data values. Since a signal processor is more suited to do the brute force, less decision intensive,

calculations required by the full search PNN compared to the more data dependent calculations of FASTPNN, than it is logical to conclude that the full search PNN will exhibit higher execution efficiency over a larger range of input data size.

Figure 12 is used to illustrate the points made in the above paragraphs. Figure 12 shows four hypothetical curves which are used to contrast the execution efficiency of FASTPNN versus full Search PNN for both a data and signal processor as a function of input data set size, N. The four curves include : (1) the execution time of FASTPNN on a signal processor as a function of N; (2) the execution time of FASTPNN on a data processor as a function of N; (3) the execution of FASTPNN on a signal processor as a function of N, and (4) the execution of full search PNN on a signal processor as a function of N. Figure 10 shows that for small values of N, for both the signal and data processor implementation, the full search implementation of PNN is more efficient than FASTPNN. But, for both the data and signal processor alike, as N gets large, FASTPNN eventually exhibits higher execution efficiency than the full search implementation of PNN. Note from Figure 12 that BREAK_EVEN_N for the data processor occurs for a much smaller input data size, than it does for the signal processor. Also note from Figure 12, that the data processor displays slightly better execution efficiency than the signal processor for all values of N in executing FASTPNN. But in executing the full search implementation of PNN, the signal processor significantly outperforms the data processor for all values of N.

FIGURE 12. HYPOTHETICAL EFFICIENCY COMPARISON OF DATA PROCESSOR VERSUS SIGNAL PROCESSOR



9. CONCLUSIONS

9.1 Ada With Checks Versus Ada Without Checks

Ada run-time checks impose a significant penalty on Ada execution efficiency for both the MIPS and the VAX. On the VAX there was a 43% relative time "penalty" (increase in time) associated with performing run-time checks, while on the MIPS, there was a 65% time penalty associated with performing run-time checks.

9.2 C Versus Ada

There is no inherit execution efficiency advantage of C over Ada or vice versa. The comparison of C versus Ada depends on the relative maturity of the compilers used. For instance, on the MIPS Magnum 3000, the C code executes more efficiently than the Ada code. While on the VAX, the Ada code executes more efficiently than the C code. Thus, the resulting conclusion is that for the MIPS Magnum 3000, the C compiler is more mature than the Ada compiler, while for the VAX 11/780 the Ada compiler is more mature than the C compiler.

9.3 MIPS Magnum 3000 Versus VAX 11/780

Depending on the language used in the comparison, the MIPS Magnum 3000 executed between 9 and 22 times more efficiently (9 to 22 times less time) than the VAX 11/780. When Ada was the language compared on both machines, the MIPS executed 9 times faster than the VAX. When C was the language, the MIPS executes 22 times faster than the VAX. The descrepancy between C and Ada indicate that the relative efficiency of MIPS C over VAX C is larger than the efficiency of MIPS Ada over VAX Ada.

9.4 FASTPNN/ 4 Real-Time Implementation Considerations

The best case FASTPNN execution time obtained from executing the C coded algorithm on the MIPS Magnum 3000 is estimated to be approximately 3 orders of magnitude too slow for real time use. Specialized signal processor hardware will be required to boost the execution efficiency of FASTPNN to real time performance levels. When using specialized signal processor hardware, it may be advantageous to implement the full search PNN algorithm. The brute force, less decision intensive calculations required by the full search PNN algorithm make it more suitable for signal processor hardware application than the FASTPNN algorithm. Since the choice of algorithm implementation is dependent on the input data set size, a study should be performed to determine which implementation is best for a given input data set size. Given equal execution times for both the full search PNN implementation and the FASTPNN implementation, the full search implementation is preferred since it provides the more accurate results.

REFERENCES

1. Cohen, Norman. Ada as a Second Lanaguage. McGraw-Hill Book Company, 1986.

2. Equitz, W.H. "A New Vector Quantization Clustering Algorithm," IEEE Transactions on Acoustics, Speech, and Signal Processing, Volume 37, number 10, October 1989.

3. Kernighan, Brian W. and Ritchie, Dennis M. The C Programming Language. Prentice Hall Software Series, 1988.

APPENDIX A

C CODED FASTPNN BENCHMARK SOURCE CODE

TABLE OF CONTENTS

```
#include "timer.h"
#include "fastpnn.h"
#include <stdio.h>

float *FastPNN (float *means, float *weights, int count,
                int dim, int ncntrds);

Main(int argc,  char *argv[])
   /*
    * Function name:
    *     main
    *
    *  Purpose:
    *     Function Main is the driver routine for the Fast Pairwise
    *     Nearest Neighbor Clustering Algorithm (FASTPNN).  This routine
    *     calls  routines to perform the timing of the FASTPNN algorithm.
    *     This routine uses command line arguments to pass parameters to
    *     itself when it begins executing.  At the command line, the user
    *     enters the program name followed by the following parameters:
    *     the input file name, the output file name, and the number of
    *     input vectors.
    */
{
   float *results;
   float *positions;
   float *weights;
   float a,b,c;
   int count;
   int dim;

   int ncntrds, i, j, k;

   int loop_count = 1;
   int initial_dummy_time, dummy_arg, dummy_elapsed_time;
   int init_fastpnn_time, fastpnn_elapsed_time;
   float fastpnn_iteration_time;
   int vector_count;
   int weight_count;
   int position_count;

   FILE *fpt_in;
   FILE *fpt_out;

   i = 0;
   j = 0;

   fpt_in = fopen(argv[1], "r");
   fpt_out = fopen(argv[2], "w");
   vector_count = atoi(argv[3], "r");
   position_count = 2 * vector_count;
   ncntrds = 4;

   positions = (float *)calloc(position_count, sizeof(float));
   weights = (float *)calloc(vector_count, sizeof(float));

   while (feof(fpt_in) == 0) {
```

```
      fscanf (fpt_in, "%f %f %f", &a, &b, &c);
      positions[i++] = a;
      positions[i++] = b;
      weights[j++] = c;
   }

   count = vector_count;
   dim = position_count/vector_count;
   printf("enter fastpnn\n");

   initial_dummy_time = init_timer();
   printf("got initial_dummy_time\n");
   for (i=0; i<loop_count; i++)
   {
      dummy_arg = identity(dummy_arg);
   }
   dummy_elapsed_time = elapsed_time(initial_dummy_time);
   printf("dummy_elapsed_time = %d", dummy_elapsed_time);

   init_fastpnn_time = init_timer();
   for (i=0; i<loop_count; i++)
   {
     results = FastPNN(positions, weights, count, dim, ncntrds);
     dummy_arg = identity(dummy_arg);
   }
   fastpnn_elapsed_time = elapsed_time(init_fastpnn_time);

   fastpnn_iteration_time = (fastpnn_elapsed_time - dummy_elapsed_time) /
                                                              loop_count;

   fprintf(fpt_out, "Fastpnn executed in %f", fastpnn_iteration_time);
   fprintf(fpt_out, "microseconds\n");
   fprintf(fpt_out, "Returned results\n");
   printarray(results, dim, ncntrds);
   fclose(fpt_out);
}

int printarray(float *array, int dim, int count)

{
   int i, j, k;

   for (i=0, k=0; i != count; i++) {
      printf("  (");
      for (j=0; j != dim; j++, k++) {
         printf("%f", array[k]);
         if (j != dim - 1)
            printf(" ");
         else
            printf(")\n");
      }
   }
}
```

```
/* This is the header file, "Fastpnn.h", which contains the
 * data structure definitions, function prototype definitions, and
 * symbolic name definitions for the FastPNN C program */

#ifndef _FastPNN_
#define _FastPNN_

#ifndef TRUE
#define TRUE -1
#endif

#ifndef FALSE
#define FALSE 0
#endif

#define KDNODE   0
#define KDBUCKET 1

#define KDMEMERR "KD tree memory allocation error\n"

#define BUCKETSIZE 8          /* Number of entries per bucket       */

#define KDMERGE 0.5           /* Fraction of buckets merged         */

#define APTR (char *)

struct kdentry {
  struct kdentry **next;      /* k dimensional linked list pointers */
  int              splitleft; /* flag used for bucket splitting     */
  float            weight;    /* Weight assigned to this entry      */
  float           *mean;      /* k dimensional sample point data    */
  float           *wmean;     /* k dimensional weighted sample data */
  float           *wsqmn;     /* k dim. weighted square sample data */
};

struct kdnode {
  int    dindx;               /* Dimension index                    */
  struct kdelem *lower;       /* Pointer to kdelems below thresh    */
  struct kdelem *upper;       /* Pointer to kdelems above thresh    */
};

struct kdbucket {
  int    count;               /* Cardinality of bucket entries      */
  struct kdentry **lists;     /* Pointers to sorted data linked list */
  struct kdentry *entrya;     /* First element of candidate pair    */
  struct kdentry *entryb;     /* Second element of candidate pair   */
  float  distort;             /* Distortion induced by merging pair */
};

struct kdelem {
  int type;                   /* value of KDNODE or KDBUCKET        */
  union {                     /* node or bucket union               */
    struct kdnode   node;
    struct kdbucket bucket;
  } norb;
};
```

```
struct kdtree {
  int             dim;       /* Dimension of tree entries          */
  struct kdelem  *root;      /* Pointer to first kd tree element   */
  int             nbuckets;  /* Number of terminal nodes           */
  int             nentries;  /* Total number of sample points      */
};

#endif

void fatal_message(char string[]);
```

```
#include "fastpnn.h"

float *FastPNN (means,weights,count,dim,ncntrds)

     float *means, *weights;
     int count, dim, ncntrds;

/*
 * Function name:
 *    FastPNN
 *
 * Purpose:
 *    Main routine for the Pairwise Nearest Neighbor clustering algorithm
 *
 * Input arguments:
 *    means     - sample point array
 *    weights   - sample weight array
 *    count     - number of samples
 *    dim       - dimensionality of the sample data
 *    ncntrds   - target number of clusters to form from the data
 *
 * Output arguments:
 *    None
 *
 * Returns:
 *    STACK     - array of clusters
 */

{
  struct kdentry *entry;
  struct kdtree  *tree, *BuildKDtree();
  float *centroids;
  void MergeDownKDtree(), DestroyKDtree();
/*  char *calloc();  */
  int i, j, k;

  if (!(centroids = (float *)calloc((unsigned)ncntrds*dim,sizeof(float))))
    fatal_message(KDMEMERR);

  tree = BuildKDtree(means,weights,count,dim);

  MergeDownKDtree(tree,ncntrds);

  entry = tree->root->norb.bucket.lists[0];
  for (i=0,k=0; i!=ncntrds; i++,entry=entry->next[0])
    for (j=0; j!=dim; j++,k++)
      centroids[k] = entry->mean[j];

  DestroyKDtree(tree);

  return(centroids);
}
```

```
#include "fastpnn.h"

struct kdtree *BuildKDtree(means,weights,count,dim)

     float *means, *weights;
     int     count, dim;

/*
 * Function name:
 *    BuildKDtree
 *
 * Purpose:
 *    Constructs an initial kD tree from the sample data
 *
 * Input arguments:
 *    means     - sample point array
 *    weights   - sample weight array
 *    count     - number of samples
 *    dim       - dimensionality of the sample data
 *
 * Output arguments:
 *    None
 *
 * Returns:
 *    STACK     - pointer to a kD tree
 */

{
  struct kdtree *tree, *CreateKDtree();
  struct kdelem *CreateFirstBucket();

  tree           = CreateKDtree(dim);
  tree->root     = CreateFirstBucket(means,weights,count,dim);
  tree->nbuckets = 1;
  tree->nentries = count;

  return(tree);
}
```

```
#include "fastpnn.h"

struct kdtree *CreateKDtree(dim)

     int dim;

/*
 * Function name:
 *    CreateKDtree
 *
 * Purpose:
 *    Allocates storage for a kD tree data structure
 *
 * Input arguments:
 *    dim       - dimensionality of the tree data
 *
 * Output arguments:
 *    None
 *
 * Returns:
 *    STACK     - pointer to a kD tree
 */

{
  struct kdtree *tree;

  if (!(tree = (struct kdtree *)calloc((unsigned)1,sizeof(struct kdtree))))
    fatal_message(KDMEMERR);

  tree->dim = dim;

  return(tree);
}
```

```
#include "fastpnn.h"

void DestroyKDtree(tree)

     struct kdtree *tree;

/*
 * Function name:
 *    DestroyKDtree
 *
 * Purpose:
 *    Destroy a kD tree
 *
 * Input arguments:
 *    tree    - pointer to the kD tree
 *
 * Output arguments:
 *    None
 *
 * Returns:
 *    Nothing
 */

{
  void DestroyLastBucket();

  if (*ree->root)
    DestroyLastBucket(tree->root);
  cfree((char *)tree);
}
```

```
#include "fastpnn.h"

struct kdelem *CreateFirstBucket (means, weights, count, dim)

     float *means, *weights;
     int count, dim;

/*
 * Function name:
 *    CreateFirstBucket
 *
 * Purpose:
 *    Create and initialize the first bucket in a kD tree
 *
 * Input arguments:
 *    means    - sample point array
 *    weights  - sample weight array
 *    count    - number of samples
 *    dim      - dimensionality of the sample data
 *
 * Output arguments:
 *    None
 *
 * Returns:
 *    STACK    - Initialized bucket
 */

{
  struct kdelem  *bucket, *CreateKDbucket();
  struct kdentry *kdptr, *lptr, *CreateKDentry();
  void SortBucket();
  int i, j, k;

  bucket                    = CreateKDbucket(dim);
  bucket->norb.bucket.count = count;

  for (i=0,k=0; i!=count; i++) {
    kdptr         = CreateKDentry(dim);
    kdptr->weight = weights[i];
    for (j=0; j!=dim; j++,k++) {
      kdptr->mean[j]  = means[k];
      kdptr->wmean[j] = kdptr->mean[j] * kdptr->weight;
      kdptr->wsqmn[j] = kdptr->mean[j] * kdptr->wmean[j];
    }
    if (!i) {
      bucket->norb.bucket.lists[0] = kdptr;
      lptr                         = kdptr;
    }
    else {
      lptr->next[0] = kdptr;
      lptr          = kdptr;
    }
  }
  SortBucket (bucket,dim);
```

```
    return(bucket);
}
```

```
#include "fastpnn.h"

struct kdelem *CreateKDbucket(dim)

     int dim;

/*
 * Function name:
 *    CreateKDBucket
 *
 * Purpose:
 *    Create a bucket for a kD tree
 *
 * Input arguments:
 *    dim       - dimensionality of the sample data
 *
 * Output arguments:
 *    None
 *
 * Returns:
 *    STACK     - Newly created bucket
 */

{
  struct kdelem *bucket;
  /* char *calloc();  */

  if (!(bucket = (struct kdelem *)calloc((unsigned)1,sizeof(struct kdelem))))
    fatal_message(KDMEMERR);
  if (!(bucket->norb.bucket.lists  = (struct kdentry **)
        calloc((unsigned)dim,sizeof(struct kdentry *))))
    fatal_message(KDMEMERR);

  bucket->type = KDBUCKET;

  return(bucket);
}
```

```
#include "fastpnn.h"

void DestroyLastBucket (bucket)

     struct kdelem *bucket;

/*
 * Function name:
 *    DestroyLastBucket
 *
 * Purpose:
 *    Destroy the last bucket in a tree
 *
 * Input arguments:
 *    bucket   - bucket to be destroyed
 *
 * Output arguments:
 *    None
 *
 * Returns:
 *    Nothing
 */
{
  void DestroyKDbucket(), DestroyKDentry();
  struct kdentry *entry, *next;

  entry = bucket->norb.bucket.lists[0];
  while (entry) {
    next = entry->next[0];
    DestroyKDentry(entry);
    entry = next;
  }
  DestroyKDbucket(bucket);
}
```

```
#include "fastpnn.h"

void DestroyKDbucket (bucket)

     struct kdelem *bucket;

/*
 * Function name:
 *    DestroyKDBucket
 *
 * Purpose:
 *    Destroy a kD tree bucket
 *
 * Input arguments:
 *    bucket   - bucket to be destroyed
 *
 * Output arguments:
 *    None
 *
 * Returns:
 *    Nothing
 */

{
  cfree((char *)bucket->norb.bucket.lists);
  cfree((char *)bucket);
}
```

```
#include "fastpnn.h"

struct kdentry *CreateKDentry (dim)

     int dim;

/*
 * Function name:
 *    CreateKDentry
 *
 * Purpose:
 *    Create a kD tree bucket entry for holding a sample point
 *
 * Input arguments:
 *    dim      - dimensionality of the sample data
 *
 * Output arguments:
 *    None
 *
 * Returns:
 *    STACK    - pointer to a bucket entry
 */

{
  struct kdentry *entry;
/*  char *calloc();  */

  if (!(entry = (struct kdentry *)calloc((unsigned)1,sizeof(struct kdentry))))
    fatal_message(KDMEMERR);
  if (!(entry->next = (struct kdentry **)
        calloc((unsigned)dim,sizeof(struct kdentry *))))
    fatal_message(KDMEMERR);
  if (!(entry->mean  = (float *)calloc((unsigned)dim,sizeof(float))))
    fatal_message(KDMEMERR);
  if (!(entry->wmean = (float *)calloc((unsigned)dim,sizeof(float))))
    fatal_message(KDMEMERR);
  if (!(entry->wsqmn = (float *)calloc((unsigned)dim,sizeof(float))))
    fatal_message(KDMEMERR);

  return(entry);
}
```

```
#include "fastpnn.h"

void DestroyKDentry (entry)

     struct kdentry *entry;

/*
 * Function name:
 *    DestroyKDentry
 *
 * Purpose:
 *    Destroy a kD tree bucket entry
 *
 * Input arguments:
 *    entry    - pointer to the entry to be destroyed
 *
 * Output arguments:
 *    None
 *
 * Returns:
 *    Nothing
 */

{
  if (entry) {
    cfree((char *)entry->next);
    cfree((char *)entry->mean);
    cfree((char *)entry->wmean);
    cfree((char *)entry->wsqmn);
    cfree((char *)entry);
  }
}
```

```
#include "fastpnn.h"

struct kdelem *CreateKDnode()

/*
 * Function name:
 *    CreateKDnode
 *
 * Purpose:
 *    Create a kD tree node
 *
 * Input arguments:
 *    Nothing
 *
 * Output arguments:
 *    None
 *
 * Returns:
 *    STACK    - pointer to the newly created kD tree node
 */

{
  struct kdelem *node;
  /*  char *calloc();   */

  if (!(node = (struct kdelem *)calloc((unsigned)1,sizeof(struct kdelem))))
    fatal_message(KDMEMERR);

  node->type = KDNODE;

  return(node);
}
```

```
#include "fastpnn.h"

void DestroyKDnode(node)

     struct kdelem *node;

/*
 * Function name:
 *    DestroyKDnode
 *
 * Purpose:
 *    Destroy a kD tree node
 *
 * Input arguments:
 *    node    - pointer to the entry to be destroyed
 *
 * Output arguments:
 *    None
 *
 * Returns:
 *    Nothing
 */

{
  if (node)
    cfree((char *)node);
}
```

```
#include "fastpnn.h"

void SortBucket (elem,dim)

     struct kdelem *elem;
     int dim;

/*
 * Function name:
 *    SortBucket
 *
 * Purpose:
 *    Sort the entries in a kD tree bucket across each dimension separately
 *
 * Input arguments:
 *    elem      - pointer to the bucket containing the entries to be sorted
 *    dim       - dimensionality of the sample data
 *
 * Output arguments:
 *    None
 *
 * Returns:
 *    Nothing
 */

{
  struct kdentry *entry, **epbffr;
  void indxx();
 /* char *calloc();  */
  int i, j, count;

  count   = elem->norb.bucket.count;

  if (!(epbffr = (struct kdentry **)calloc((unsigned)count,
                                           sizeof(struct kdentry *))))
    fatal_message(KDMEMERR);

  for (i=0; i!=dim; i++) {
    for (j=0,entry=elem->norb.bucket.lists[0]; j!=count; j++) {
      epbffr[j] = entry;
      entry     = entry->next[0];
    }
    indxx(epbffr,&i);
    elem->norb.bucket.lists[i] = epbffr[0];
    for (j=1; j<count; j++)
      epbffr[j-1]->next[i] = epbffr[j];
    epbffr[[count - 1]]->next[i] = (struct kdentry *)0;
  }
  cfree((char *  )bffr);
}
```

```
#include "fastpnn.h"

void indxx(kdentry **epbfrr,int *i)

/*
 * Function name:
 *    indxx
 *
 * Purpose:
 *    Sort an array of indices indx based on the data arrin using an indexed
 *    version of a heap sort. Modified from Numerical Recipes indexx.
 *
 * Input arguments:
 *    arrin     - array of data used for sorting
 *    n         - number of entries in arrin
 *
 * Output arguments:
 *    indx      - Indices specifiying the order of data in arrin
 *
 * Returns:
 *    Nothing
 */

{
  int l,j,ir,indxt,i;
  float q;

  for (j=0;j<n;j++)
    indx[j] = j;

  l  = n >> 1;
  ir = n - 1;

  while (TRUE) {
    if (l > 0)
      q = arrin[(indxt=indx[--l])];
    else {
      q = arrin[(indxt=indx[ir])];
      indx[ir]=indx[0];
      if (--ir == 0) {
        indx[0]=indxt;
        return;
      }
    }

    i = l;
    j = ((l + 1)<< 1) - 1;

    while (j <= ir) {
      if (j < ir && arrin[indx[j]] < arrin[indx[j+1]])
        j++;
      if (q < arrin[indx[j]]) {
        indx[i]=indx[j];
        j += ((i=j) + 1);
      }
```

```
            else j=ir+1;
          }
          indx[i]=indxt;
        }
      }
```

```
#include "fastpnn.h"

void MergeDownKDtree(tree,ncntrds)

     struct kdtree *tree;
     int ncntrds;

/*
 * Function name:
 *    MergeDownKDtree
 *
 * Purpose:
 *    Reduce a kD tree to a single bucket having ncntrds entries using the PNN
 *    algorithm
 *
 * Input arguments:
 *    tree     - pointer to a kD tree
 *    ncntrds  - desired number of entries after merging
 *
 * Output arguments:
 *    None
 *
 * Returns:
 *    Nothing
 */

{
  void CompressKDtree(), BalanceKDtree();
  struct kdelem *CollapseKDnode();
  int nmerge, ntile, maxbkts;
  struct kdelem **bcktarray;
  /* char *calloc(); */

  maxbkts = tree->nentries / (BUCKETSIZE / 2);

  if (!(bcktarray = (struct kdelem **)calloc((unsigned)maxbkts,
                                        sizeof(struct kdelem *))))
    fatal_message(KDMEMERR);

  while (tree->nentries > ncntrds) {
    BalanceKDtree(tree,bcktarray);
    ntile  = (KDMERGE * tree->nbuckets > 1) ? KDMERGE * tree->nbuckets : 1;
    nmerge = ((tree->nentries - ncntrds) < ntile) ?
                                    (tree->nentries - ncntrds) : ntile;
    CompressKDtree(tree,nmerge,bcktarray);
  }
  if (tree->root->type == KDNODE)
    tree->root = CollapseKDnode(tree->root,tree->dim,&tree->nbuckets);

  cfree((char *)bcktarray);
}
```

```
#include "fastpnn.h"

void BalanceKDtree(tree,bcktarray)

     struct kdtree *tree;
     struct kdelem **bcktarray;

/*
 * Function name:
 *    BalanceKDtree
 *
 * Purpose:
 *    Redistribute the entries in a kD tree so that each bucket has
 *    approximately the same number of entries
 *
 * Input arguments:
 *    tree      - pointer to a kD tree
 *
 * Output arguments:
 *    bcktarray - array used to retain a pointer to each bucket after balancing
 *
 * Returns:
 *    Nothing
 */

{
  struct kdelem *CollapseKDnode(), *SplitBucket(), **bptr;
  float *mean, *wvar;
  /* char *calloc(); */

  if (!(mean = (float *)calloc((unsigned)tree->dim,sizeof(float))))
    fatal_message(KDMEMERR);
  if (!(wvar = (float *)calloc((unsigned)tree->dim,sizeof(float))))
    fatal_message(KDMEMERR);

  if (tree->root->type == KDNODE)
    tree->root = CollapseKDnode(tree->root,tree->dim,&tree->nbuckets);

  bcktarray[0] = tree->root;
  bptr         = &bcktarray[1];

  tree->root =
            SplitBucket(tree->root,tree->dim,&tree->nbuckets,&bptr,mean,wvar);

  cfree((char *)mean);
  cfree((char *)wvar);
}
```

```
#include "fastpnn.h"

struct kdelem *CollapseKDnode(elem,dim,bctr)

     struct kdelem *elem;
     int dim, *bctr;

/*
 * Function name:
 *    CollapseKDnode
 *
 * Purpose:
 *    Recursive function used to collapse the entries in a kD tree into a
 *    single bucket
 *
 * Input arguments:
 *    elem      - pointer to a kD tree node to be collapsed
 *    dim       - dimension of the data within the tree
 *
 * Output arguments:
 *    bctr      - pointer to a counter used to keep track of the total number
 *                of buckets in the tree
 *
 * Returns:
 *    STACK     - pointer to the bucket resulting from the collapse
 */

{
  struct kdentry **entptr, *lentry, *rentry;
  struct kdelem *bucket;
  void DestroyKDnode(), DestroyKDbucket();
  int i;

  if (elem->norb.node.lower->type != KDBUCKET)
    elem->norb.node.lower = CollapseKDnode(elem->norb.node.lower,dim,bctr);
  if (elem->norb.node.upper->type != KDBUCKET)
    elem->norb.node.upper = CollapseKDnode(elem->norb.node.upper,dim,bctr);

  bucket = elem->norb.node.lower;

  for (i=0; i!=dim; i++) {
    entptr = &(elem->norb.node.lower->norb.bucket.lists[i]);
    lentry =   elem->norb.node.lower->norb.bucket.lists[i];
    rentry =   elem->norb.node.upper->norb.bucket.lists[i];
    while (lentry && rentry) {
      if (lentry->mean[i] < rentry->mean[i]) {
        *entptr = lentry;
         entptr = &(lentry->next[i]);
         lentry = lentry->next[i];
      }
      else {
        *entptr = rentry;
         entptr = &(rentry->next[i]);
         rentry = rentry->next[i];
      }
```

```
    }
    if (lentry) {
      while (lentry) {
        *entptr = lentry;
         entptr = &(lentry->next[i]);
         lentry = lentry->next[i];
      }
    }
    if (rentry) {
      while (rentry) {
        *entptr = rentry;
         entptr = &(rentry->next[i]);
         rentry = rentry->next[i];
      }
    }
  }
  bucket->norb.bucket.count = elem->norb.node.lower->norb.bucket.count +
                              elem->norb.node.upper->norb.bucket.count;
  DestroyKDbucket(elem->norb.node.upper);
  DestroyKDnode  (elem);
  (*bctr) --;
  return(bucket);
}
```

```
#include "fastpnn.h"

struct kdelem *SplitBucket(oldbucket,dim,bctr,bptr,mean,wvar)

     struct kdelem *oldbucket, ***bptr;
     int dim, *bctr;
     float   *mean, *wvar;

/*
 * Function name:
 *    SplitBucket
 *
 * Purpose:
 *    Recursive function used to split a kD tree bucket into two smaller
 *    buckets having half as many entries
 *
 * Input arguments:
 *    oldbucket - pointer to kD tree bucket to be split
 *    dim       - dimension of the data within the tree
 *    mean      - scratch array used for calculating bucket means
 *    wvar      - scratch array used for calculating bucket weighted variances
 *
 * Output arguments:
 *    bctr      - pointer to a counter used to keep track of the total number
 *                of buckets in the tree
 *    bptr      - pointer to an array of pointers to buckets in the tree
 *
 * Returns:
 *    STACK     - pointer to the node resulting from the split, or the original
 *                bucket if the number of entries in the bucket is small enough
 */

{
  struct kdelem *newnode, *newbucket, *CreateKDbucket();
  struct kdentry **oldptr, **newptr, *entry;
  int i, j, bcount, medindx;
  void GetBucketStats();

  if (oldbucket->norb.bucket.count > BUCKETSIZE) {

    GetBucketStats(oldbucket,dim,mean,wvar);

    for (i=1,j=0; i<dim; i++)
      if (wvar[i] > wvar[j])
        j = i;

    bcount  = oldbucket->norb.bucket.count;
    medindx = (bcount + 1) / 2;              /* Uneven splits go left */

    newnode                  = CreateKDnode();
    newnode->norb.node.dindx = j;

    newbucket = CreateKDbucket(dim);

    newnode->norb.node.lower = oldbucket;
```

```
    newnode->norb.node.upper = newbucket;

    ( *bctr) ++;
    (**bptr) = newbucket;
    ( *bptr) ++;

    for (i=0, entry=oldbucket->norb.bucket.lists[j]; i<medindx; i++) {
      entry->splitleft = TRUE;
      entry            = entry->next[j];
    }

    for (i=medindx; i<bcount; i++) {
      entry->splitleft = FALSE;
      entry            = entry->next[j];
    }

    oldbucket->norb.bucket.count = medindx;
    newbucket->norb.bucket.count = bcount - medindx;

    for (i=0; i!=dim; i++) {
      oldptr = &oldbucket->norb.bucket.lists[i];
      newptr = &newbucket->norb.bucket.lists[i];
      entry  =  oldbucket->norb.bucket.lists[i];
      while (entry) {
        if (entry->splitleft) {
          *oldptr = entry;
           oldptr = (struct kdentry **)(&entry->next[i]);
        }
        else {
          *newptr = entry;
           newptr = (struct kdentry **)(&entry->next[i]);
        }
        entry = entry->next[i];
      }
      *oldptr = (struct kdentry *)0;
      *newptr = (struct kdentry *)0;
    }

    newnode->norb.node.lower =
                SplitBucket(newnode->norb.node.lower,dim,bctr,bptr,mean,wvar);
    newnode->norb.node.upper =
                SplitBucket(newnode->norb.node.upper,dim,bctr,bptr,mean,wvar);

    return(newnode);
  }
  else
    return(oldbucket);
}
```

```
#include "fastpnn.h"

void GetBucketStats (elem,dim,mean,wvar)

     struct kdelem *elem;
     int dim;
     float *mean, *wvar;

/*
 * Function name:
 *    GetBucketStats
 *
 * Purpose:
 *    Calculate the k dimensional means and weighted variances for a bucket
 *
 * Input arguments:
 *    elem      - pointer to the kD tree bucket for which the statistics are
 *                to be calculated
 *    dim       - dimension of the data within the tree
 *
 * Output arguments:
 *    mean      - array used for calculating bucket means
 *    wvar      - array used for calculating bucket weighted variances
 *
 * Returns:
 *    Nothing
 */

{
  struct kdentry *entry;
  float wgtsum;
  int i;

  wgtsum = 0.0;

  for (i=0; i!=dim; i++) {
    mean[i] = 0.0;
    wvar[i] = 0.0;
  }

  entry = elem->norb.bucket.lists[0];
  while (entry) {
    wgtsum += entry->weight;
    for (i=0; i!=dim; i++) {
      mean[i] += entry->wmean[i];
      wvar[i] += entry->wsqmn[i];
    }
    entry = entry->next[0];
  }

  for (i=0; i!=dim; i++) {
    mean[i] /= wgtsum;
    wvar[i]  = (wvar[i] / wgtsum) - (mean[i] * mean[i]);
  }
}
```

```
#include "fastpnn.h"

static int BucketCompare(one,two)

     struct kdelem **one, **two;

/*
 * Function name:
 *    BucketCompare
 *
 * Purpose:
 *    Function used by the UNIX qsort routine to compare merge distortions of
 *    two buckets
 *
 * Input arguments:
 *    one       - pointer to a pointer to the first kD tree bucket used in
 *                the comparison
 *    two       - pointer to a pointer to the second kD tree bucket used in
 *                the comparison
 *
 * Output arguments:
 *    None
 *
 * Returns:
 *    STACK     - the value  1 if distortion(one) > distortion(two)
 *                the value  0 if distortion(one) = distortion(two)
 *                the value -1 if distortion(one) < distortion(two)
 */

{
  if      ((*one)->norb.bucket.distort  < (*two)->norb.bucket.distort)
    return(-1);
  else if ((*one)->norb.bucket.distort == (*two)->norb.bucket.distort)
    return( 0);
  else
    return( 1);
}
```

```
#include "fastpnn.h"

void CompressKDtree (tree, nmerge, bcktarray)

     struct kdtree *tree;
     int nmerge;
     struct kdelem **bcktarray;

/*
 * Function name:
 *    CompressKDtree
 *
 * Purpose:
 *    Function used to merge bucket entry pairs into single bucket entries for
 *    a fixed fraction of the total number of buckets
 *
 * Input arguments:
 *    tree      - pointer to the kD tree undergoing the merge
 *    nmerge    - number of bucket pairs to merge
 *    bcktarray - array of pointers to all buckets in the tree
 *
 * Output arguments:
 *    None
 *
 * Returns:
 *    Nothing
 *
 */

{
  void AssessCandidate(), ReduceKDbucket();
  int i, ncount, BucketCompare();

  for (i=0,ncount=0; i!=tree->nbuckets; i++)
    AssessCandidate(bcktarray[i],tree->dim,bcktarray,&ncount);

  /* Handle end game situations */

  if (nmerge > ncount)
    nmerge = ncount;

  if (ncount > 1)
    qsort((char *)bcktarray,ncount,sizeof(struct kdelem *),BucketCompare);

  for (i=0; i!=nmerge; i++)
    ReduceKDbucket(bcktarray[i],tree->dim);

  tree->nentries -= nmerge;
}
```

```
#include "fastpnn.h"

void AssessCandidate(elem,dim,barray,ncount)

     struct kdelem *elem, **barray;
     int dim, *ncount;

/*
 * Function name:
 *    AssessCandidate
 *
 * Purpose:
 *    Determine the minimal distortion than can be produced by merging a pair
 *    of bucket entries
 *
 * Input arguments:
 *    elem      - pointer to the kD tree bucket under evaluation
 *    dim       - dimension of the data within the entries
 *
 * Output arguments:
 *    barray    - array of pointers to all buckets that can be merged
 *    ncount    - pointer to counter used to keep track of the total number
 *                of buckets that can be merged
 *
 * Returns:
 *    Nothing
 *
 */

{
  struct kdentry *ientry, *jentry;
  float reduction, dotprd, diff;
  int i, j, k, firsttime;

  firsttime = TRUE;
  if (elem->norb.bucket.count > 1) {
    ientry = elem->norb.bucket.lists[0];
    for (i=0; i<elem->norb.bucket.count-1; i++,ientry=ientry->next[0]) {
      jentry = ientry->next[0];
      for (j=i+1; j<elem->norb.bucket.count; j++,jentry=jentry->next[0]) {
        for (k=0,dotprd=0.0; k<dim; k++) {
          diff    = ientry->mean[k] - jentry->mean[k];
          dotprd += diff * diff;
        }
        reduction = dotprd * ientry->weight * jentry->weight /
          (ientry->weight + jentry->weight);
        if ((reduction < elem->norb.bucket.distort) || firsttime) {
          elem->norb.bucket.distort = reduction;
          elem->norb.bucket.entrya  = ientry;
          elem->norb.bucket.entryb  = jentry;
          firsttime                 = FALSE;
        }
      }
    }
    barray[(*ncount)++] = elem;
```

```
    }
}
```

```
#include "fastpnn.h"

void ReduceKDbucket(elem,dim)

     struct kdelem *elem;
     int dim;

/*
 * Function name:
 *    ReduceKDbucket
 *
 * Purpose:
 *    Merges a pair of bucket entries into a single entry
 *
 * Input arguments:
 *    elem      - pointer to the kD tree bucket whose entries are to be merged
 *    dim       - dimension of the data within the entries
 *
 * Output arguments:
 *    None
 *
 * Returns:
 *    Nothing
 *
 */

 {
   struct kdentry *ientry, *jentry, **leptr;
   float newweight;
   int i, rmcnt;

   ientry = elem->norb.bucket.entrya;
   jentry = elem->norb.bucket.entryb;

   /* Remove ientry, jentry from the list */

   for (i=0; i!=dim; i++) {
     for (leptr = &(elem->norb.bucket.lists[i]),rmcnt=0; rmcnt!=2;) {
       if ((*leptr==jentry) || (*leptr==ientry)) {
         *leptr = (*leptr)->next[i];
         rmcnt++;
       }
       else
          leptr = &((*leptr)->next[i]);
     }
   }

   newweight = ientry->weight + jentry->weight;
   for (i=0; i!=dim; i++) {
     ientry->mean[i] = (ientry->mean[i] * ientry->weight  +
                        jentry->mean[i] * jentry->weight) / newweight;
     ientry->wmean[i] = ientry->mean[i] * newweight;
     ientry->wsqmn[i] = ientry->mean[i] * ientry->wmean[i];
   }
   ientry->weight = newweight;
```

```
  /* Reinsert ientry into the list in the proper order */

  for (i=0; i!=dim; i++) {
    for (leptr = &(elem->norb.bucket.lists[i]);
        (*leptr && ((*leptr)->mean[i] < ientry->mean[i]));
         leptr = &((*leptr)->next[i]));
    ientry->next[i] = *leptr;
    *leptr          = ientry;
  }

  elem->norb.bucket.count--;  /* Decrement the total entry count */
  DestroyKDentry(jentry);     /* Free up jentry memory           */
}


void fatal_message(char string[])
{
   printf("%s\n",string);

   return;
}
```

```
/* This is the header file,"timer.h", which contains the declarations
 * (or function prototypes) for the VAX/VMS timing routines
 *
*/

 int init_timer(void);

 int elapsed_time(int starting_time);

 int identity (int arg);
```

```
#include "timer.h"
#include <time.h>
#include <stdio.h>

/*  This file contains the VAX/VMS timing routines. Note that the
 *  functions init_timer and elapsed_time make use of a predefined
 *  C function called "times". Function times returns the accumulated
 *  CPU time in a predefined time structure called tbuffer_t.
 *
*/

tbuffer_t *init_time_ptr, *final_time_ptr;
tbuffer_t buffer, init_time, final_time;

   int init_timer (void)
   {
      int current_time;
      init_time_ptr = &init_time;
      times(init_time_ptr);
      current_time = (init_time_ptr -> proc_user_time) * 10000;
      return current_time;
   }


   int elapsed_time(int start_time)
   {
      int time_elapsed;
      final_time_ptr = &final_time;
      times(final_time_ptr);
      time_elapsed = ((final_time_ptr->proc_user_time) * 10000) - start_time;
      return time_elapsed;
   }

   int identity (int arg)
   {
       int some_value = 0;
       some_value = some_value + arg;
       return some_value;
   }
```

```
/*  This is the header file, "timer.h", which contains the declarations
 *  (or function prototypes) for the MIPS/UNIX timing routines.
*/


    long int init_timer(void);

    long int elapsed_time(void);

    int identity (int arg);
```

```
#include "timer.h"
#include <time.h>
#include <stdio.h>

/*  This file contains the MIPS/UNIX timing routines. Note that the
 *  functions init_timer and elapsed_time make use of the predefined
 *  C function called "clock".  Function clock returns the amount of
 *  CPU time used since the first call to clock.
 *
 */

long int init_timer (void)
{
   long int current_time;
   current_time = clock();
   return current_time;
}


long int elapsed_time(void)
{
   long int time_elapsed;
   time_elapsed = clock();
   return time_elapsed;
}

int identity (int arg)
{
    int some_value = 0;
    some_value = some_value + arg;
    return some_value;
}
```

58.000000 19.000000 62.000000
59.000000 19.000000 74.000000
60.000000 19.000000 70.000000
57.000000 20.000000 76.000000
58.000000 20.000000 90.000000
59.000000 20.000000 91.000000
60.000000 20.000000 78.000000
61.000000 20.000000 52.000000
56.000000 21.000000 75.000000
57.000000 21.000000 96.000000
58.000000 21.000000 101.000000
59.000000 21.000000 91.000000
60.000000 21.000000 67.000000
56.000000 22.000000 75.000000
57.000000 22.000000 100.000000
58.000000 22.000000 108.000000
59.000000 22.000000 104.000000
60.000000 22.000000 92.000000
61.000000 22.000000 72.000000
56.000000 23.000000 84.000000
57.000000 23.000000 135.000000
58.000000 23.000000 159.000000
59.000000 23.000000 165.000000
60.000000 23.000000 155.000000
61.000000 23.000000 124.000000
62.000000 23.000000 61.000000
56.000000 24.000000 105.000000
57.000000 24.000000 164.000000
58.000000 24.000000 190.000000
59.000000 24.000000 196.000000
60.000000 24.000000 185.000000
61.000000 24.000000 151.000000
62.000000 24.000000 74.000000
55.000000 25.000000 59.000000
56.000000 25.000000 109.000000
57.000000 25.000000 173.000000
58.000000 25.000000 200.000000
59.000000 25.000000 206.000000
60.000000 25.000000 194.000000
61.000000 25.000000 160.000000
62.000000 25.000000 81.000000
63.000000 25.000000 58.000000
36.000000 26.000000 52.000000
56.000000 26.000000 105.000000
57.000000 26.000000 165.000000
58.000000 26.000000 191.000000
59.000000 26.000000 197.000000
60.000000 26.000000 185.000000
61.000000 26.000000 152.000000
62.000000 26.000000 80.000000
63.000000 26.000000 58.000000
36.000000 27.000000 60.000000
37.000000 27.000000 73.000000
40.000000 27.000000 85.000000
41.000000 27.000000 94.000000

APPENDIX B

ADA CODED FASTPNN BENCHMARK SOURCE CODE

TABLE OF CONTENTS

```
with timer;
with FASTPNN;
with READ_DATA;
with text_io; use text_io;
with Data_Struct_Pkg; use Data_Struct_Pkg;
procedure main is

-- ------------------------------------------------------------
-- Procedure Name:  Main
--
-- Purpose :  This is the main routine for the FASTPNN algorithm.
--            The user is prompted to input the name of the input
--            file, the name of the output file, and the number of
--            input vector data values.  External procedure
--            READ_DATA is called by Main to read in the values
--            from the input data file.  Main performs the timing
--            of the FastPNN algorithm (by calling timing
--            routines contained within Timer_Pkg) and outputs the
--            results.
-- ------------------------------------------------------------

    LENGTHIN : INTEGER;
    LENGTHOUT : INTEGER;
    POSITIONS_LAST : INTEGER;
    RESULTS_LAST : INTEGER;
    WEIGHTS_LAST : INTEGER;
    count : integer;
    dim   : integer;
    ncntrds : integer;
    INNAME : STRING(1 .. 80);
    OUTNAME : STRING(1 .. 80);
    VECTOR_NUM : INTEGER;

    loop_count : constant := 1;
    fastpnn_elapsed : integer;
    fastpnn_timer : timer.microsec_timer;

    dummy_timer : timer.microsec_timer;
    dummy_elapsed_time : integer;
    dummy_arg : integer;

    INFILE  : FILE_TYPE;
    OUTFILE : FILE_TYPE;

    package float_io is new text_io.float_io(float);
    package int_io is new text_io.integer_io(integer);

    procedure printarray (OUTFILE : in out FILE_TYPE;
                          arrayy : in means_array_type;
                          dim : in integer; count : in integer) is

    k : integer;
    package float_io is new text_io.float_io(float);

    begin
        k := 0;
```

```
            for i in 0 .. count - 1 loop
                put(OUTFILE, "   (");
                for j in 0 .. dim - 1 loop
                    float_io.put(OUTFILE, arrayy(k));
                    k := k + 1;
                    if (j /= dim-1) then
                       put(OUTFILE, "  ");
                    else
                       put_line(OUTFILE, ")");
                    end if;
                end loop;
            end loop;
        end printarray;

begin  -- MAIN

        PUT_LINE("Enter Name of Input File. ");
        GET_LINE(INNAME, LENGTHIN);
        PUT_LINE("Enter Name of Output File ");
        GET_LINE(OUTNAME, LENGTHOUT);
        PUT_LINE("Enter the number of input vectors ");
        INT_IO.GET(VECTOR_NUM);
        OPEN(INFILE, IN_FILE, INNAME(1 .. LENGTHIN));
        CREATE(OUTFILE, OUT_FILE, OUTNAME(1 .. LENGTHOUT));

        POSITIONS_LAST := 2 * VECTOR_NUM - 1;
        RESULTS_LAST  := 2 * VECTOR_NUM - 1;
        WEIGHTS_LAST  := VECTOR_NUM - 1;

        DECLARE

          positions : means_array_type(0 .. POSITIONS_LAST);

          results : means_array_type(0 .. RESULTS_LAST);

          weights : weights_array_type(0 .. WEIGHTS_LAST);

        begin

          count :=  weights'length;
          dim   :=  positions'length/count;

          READ_DATA (INFILE, positions, weights);

          ncntrds := 4;  -- number of desired output vectors

          -- first time the dummy loop
          --
          timer.init_timer(dummy_timer);
          for i in 1 .. loop_count loop
              dummy_arg := timer.identity(dummy_arg);
          end loop;
          if timer.always_true then
              dummy_elapsed_time := timer.elapsed_time(dummy_timer);
          end if;
```

```
        --
        -- now time the routine of interest
        --

        timer.init_timer(fastpnn_timer);
        for i in 1 .. loop_count loop
            FastPNN(positions, weights, count, dim, ncntrds, results);
            dummy_arg := timer.identity(dummy_arg);
        end loop;

        if timer.always_true then
            fastpnn_elapsed := timer.elapsed_time(fastpnn_timer);
        end if;

        NEW_LINE(OUTFILE);

      --
      -- now subtract the dummy (overhead) loop and report the results
      --

      PUT(OUTFILE, "FastPNN executed in ");
      FLOAT_IO.PUT(OUTFILE, float((fastpnn_elapsed
                        dummy_elapsed_time)) / float(loop_count));

      text_io.put_line (OUTFILE, "  microseconds.");

      PUT_LINE(OUTFILE, "Returned results");

      printarray (OUTFILE, results, dim, ncntrds);

      CLOSE(OUTFILE);

      end;

end main;
```

```
with Data_Struct_Pkg; use Data_Struct_Pkg;
with Build_Pkg;
with MergeDown_Pkg;
with Destroy_Pkg;

procedure FastPNN (means : in out means_array_type;
                   weights : in out weights_array_type;
                   count : in integer;
                   dim : in integer;
                   ncntrds : in integer;
                   centroids : out means_array_type) is
 -- ------------------------------------------------------------
 --
 --  Procedure name: FastPNN
 --
 --  Purpose: Outermost routine for the Pairwise Nearest Neighbor
 --           Clustering algorithm
 --
 --  Input arguments:
 --     means   - sample point array
 --     weights - sample weight array
 --     count   - number of samples
 --     dim     - dimensionality of the sample data
 --     ncntrds - target number of clusters to form the data
 --
 --  Output arguments
 --     centroids - cluster vector results from FASTPNN algorithm
 --
 -- ------------------------------------------------------------
entryy : kdentry_ptr_type;
tree : kdtree_ptr_type;
k : integer;

begin

    Build_Pkg.BuildKDtree(means, weights, count, dim, tree);
    MergeDown_Pkg.MergeDownKDtree(tree,ncntrds);

    entryy := tree.root.bucket.lists_array(0);
    k := 0;
    for i in 0 .. ncntrds - 1 loop
        for j in 0 .. dim - 1 loop
            centroids(k) := entryy.mean_array(j);
            k := k + 1;
        end loop;
        entryy := entryy.next_array(0);
    end loop;

    Destroy_Pkg.DestroyKDtree(tree);

end FastPNN;
```

```
package Data_Struct_Pkg is

   -- ------------------------------------------------------
   -- This package specification contains the declarations of
   -- data types used by all modules in the FASTPNN routine
   -- ------------------------------------------------------

   TRUEE       : constant := -1;
   FALSEE      : constant := 0;
   KDNODEE     : constant := 0;
   KDBUCKETT   : constant := 1;
   BUCKETSIZE  : constant := 8;
   KDMERGE     : constant := 0.5;

   dim : constant := 2;

   -- Data Structures defined in the main program

   type means_array_type is array (integer range <>) of float;

   type weights_array_type is array (integer range <>) of float;

   -- data structure used in CollapseKDnode
   type integer_ptr_type is access integer;

   -- data structures needed in indxx
   type float_array_type is array (integer range <>) of float;

   type integer_array_type is array (integer range <>) of integer;

   type Kdentry;
   type Kdentry_ptr_type is access Kdentry;
   type Kdentry_ptr_array_type is array (integer range <>)
                                          of Kdentry_ptr_type;
   type mean_array_type is array (integer range <>) of float;
   type wmean_array_type is array (integer range <>) of float;
   type wsqmn_array_type is array (integer range <>) of float;
   type wvar_array_type is array (integer range <>) of float;

   type Kdentry is
      record
         next_array  : Kdentry_ptr_array_type(0 .. dim - 1);
         splitleft   : integer;
         weight      : float;
         mean_array  : mean_array_type(0 .. dim - 1);
         wmean_array : wmean_array_type(0 .. dim - 1);
         wsqmn_array : wsqmn_array_type(0 .. dim - 1);
      end record;

   type data_structure_type is (kdnode_type, kdbucket_type);
   type kdelem(data_structure : data_structure_type);
   type kdelem_ptr_type is access kdelem;


   -- Data structure kdelem_ptr_array_type used in BalanceKDtree
   type kdelem_ptr_array_type is array (integer range <>)
```

```
                                                    of kdelem_ptr_type;

    type kdnode is
        record
            dindx       : integer;
            lower       : kdelem_ptr_type;
            upper       : kdelem_ptr_type;
        end record;

    type kdbucket is
        record
            count       : integer;
            lists_array : Kdentry_ptr_array_type(0 .. dim - 1);
            entrya      : Kdentry_ptr_type;
            entryb      : Kdentry_ptr_type;
            distort     : float;
        end record;

    type kdbucket_ptr_type is access kdbucket;

    type kdelem (data_structure : data_structure_type) is
        record
            typee        : integer;
            case data_structure is
                when kdnode_type =>
                    node        : kdnode;
                when kdbucket_type =>
                    bucket      : kdbucket;
            end case;
        end record;
        norb_n : kdelem(kdnode_type);
        norb_b : kdelem(kdbucket_type);

    type kdtree is
        record
            dim         : integer;
            root        : kdelem_ptr_type;
            nbuckets    : integer;
            nentries    : integer;
        end record;

    type kdtree_ptr_type is access kdtree;

end Data_Struct_Pkg;
```

```
with Data_Struct_Pkg; use Data_Struct_Pkg;
package Build_Pkg is

    -- ------------------------------------------------------------
    --  Package Specification Build_Pkg contains the declaration of
    --  all subprograms which are available for building and
    --  initialization of the K-d data structure.
    -- ------------------ -----------------------------------------

    procedure BuildKDtree (means : in out means_array_type;
                           weights : in out weights_array_type;
                           count : in integer;
                           dim   : in integer;
                           tree : out Kdtree_ptr_type);

    function CreateKDbucket (dim : integer) return kdelem_ptr_type;

end Build_Pkg;
```

```
package body Build_Pkg is

    function CreateKDbucket (dim : integer) return kdelem_ptr_type
                                                        is separate;

    procedure Indxx (arrin : in float_array_type;
                     indx : out integer_array_type;
                     n  : in integer) is separate;

    procedure SortBucket (elem : in kdelem_ptr_type;
                          dim : in integer) is separate;

    function CreateKDentry (dim : in integer) return kdentry_ptr_type
                                                        is separate;

    procedure CreateFirstBucket (means : in out means_array_type;
                                 weights : in out weights_array_type;
                                 count : in integer;
                                  dim  : in integer;
                                 bucket : out kdelem_ptr_type)
                                                        is separate;

    function CreateKDtree (dim : integer) return kdtree_ptr_type
                                                        is separate;

    procedure BuildKDtree (means : in out means_array_type;
                           weights : in out weights_array_type;
                           count : in integer;
                           dim   : in integer;
                           tree : out Kdtree_ptr_type) is separate;

end Build_Pkg;
```

```
separate (Build_Pkg)
procedure BuildKDtree (means : in out means_array_type;
                       weights : in out weights_array_type;
                       count : in integer;
                       dim : in integer;
                       tree : out  Kdtree_ptr_type) is

-- ------------------------------------------------------------------
--  Procedure name: Buildkdtree

--  Purpose: Constructs an initial KD tree from the sample data

--  Input arguments:
--      means   -  sample point array
--      weights -  sample weight array
--      count   -  number of samples
--      dim     -  dimensionality of the sample data

--  Output arguments:
--      pointer to a kdtree

-- ------------------------------------------------------------------

    temp_tree : kdtree_ptr_type;
    bucket : kdelem_ptr_type;

begin  -- BuildKDtree

    temp_tree := CreateKDtree (dim);
    CreateFirstBucket (means, weights, count, dim, bucket);
    temp_tree.root := bucket;
    temp_tree.nbuckets := 1;
    temp_tree.nentries := count;
    tree := temp_tree;
end BuildKDtree;
```

```
separate (Build_Pkg)
procedure CreateFirstBucket (means : in out means_array_type;
                             weights : in out weights_array_type;
                             count : in integer;
                             dim : in integer;
                             bucket :  out kdelem_ptr_type) is

-- ------------------------------------------------------------------------
-- Procedure name:
--    CreateFirstBucket

-- Purpose:
--    Create and initialize the first bucket in a KD tree

-- Input arguments:
--    means   -  sample point array
--    weights -  sample weight array
--    count   -  number of samples
--    dim     -  dimensionality of the sample data

-- Output arguments:
--    bucket  -  pointer to first bucket in the Kdtree
-- ------------------------------------------------------------------------

    mean, wvar  : mean_array_type(0 .. dim -1);
    kdptr, lptr : kdentry_ptr_type;
    KDPTR0, KDPTR1 : KDENTRY_PTR_TYPE;
    k : integer;
    tempbucket : Kdelem_ptr_type;

begin  -- CreateFirstBucket

    tempbucket := CreateKDbucket(dim);
    tempbucket.bucket.count := count;

    k := 0;
    for  i in 0 .. (count - 1)  loop
        kdptr := CreateKDentry(dim);
        kdptr.weight := weights(i);
        for j in 0 .. (dim - 1) loop
            kdptr.mean_array(j) := means(k);
            kdptr.vmean_array(j) := kdptr.mean_array(j) * kdptr.weight;
            kdptr.vsqmn_array(j) := kdptr.mean_array(j) *
                                                  kdptr.wmean_array(j);
            k := k + 1;
         end loop;
         if (i=0) then
             tempbucket.bucket.lists_array(0) := kdptr;
             lptr := kdptr;
         else
             lptr.next_array(0) := kdptr;
             lptr := kdptr;
         end if;
    end loop;

     SortBucket (tempbucket, dim);
```

```
    bucket := tempBucket;
end CreateFirstBucket;
```

```
with Data_Struct_Pkg; use Data_Struct_Pkg;
separate (Build_Pkg)
function CreateKDbucket (dim : integer) return kdelem_ptr_type is

-- ------------------------------------------------------------------
-- Function name:
--    CreateKDbucket

-- Purpose:
--    Create a bucket for a KD tree

-- Input arguments
--    dim - dimensionality of the sample data

-- Output arguments
--    None

-- Returns:
--    returns pointer to new created bucket

-- ------------------------------------------------------------------

    bucket : kdelem_ptr_type;

begin

    bucket := new kdelem(kdbucket_type);
    bucket.typee := KDBUCKETT;
    return bucket;

end CreateKDbucket;
```

```
separate (Build_Pkg)
function CreateKDentry (dim : in integer) return kdentry_ptr_type is

-- ------------------------------------------------------------------

--  Function name:
--      CreateKDentry

--  Purpose:
--      Create a KD tree bucket entry for holding a sample point

--  Input arguments:
--      dim  - dimensionality of the sample data

--  Output arguments:
--      None

--  Returns:
--      pointer to a bucket entry
-- ------------------------------------------------------------------

    entryy : kdentry_ptr_type;

begin

    entryy := new Kdentry;
    return entryy;

end CreateKDentry;
```

```
separate (Build_Pkg)
function CreateKDtree (dim : integer) return kdtree_ptr_type is

-- ------------------------------------------------------------
-- Function name:
--     CreateKDtree

-- Purpose:
--    dynamically allocates storage space for a KD tree data structure

-- Input Arguments:
--    dim  - dimensionality of the tree data

-- Output arguments
--    None

-- Returns:
--    pointer to a KDtree
-- ------------------------------------------------------------

    tree : kdtree_ptr_type;

begin

    tree := new kdtree;
    tree.dim := dim;
    return tree;

end CreateKDtree;
```

```
separate (Build_Pkg)
procedure Indxx (arrin : in float_array_type;
                 indx :  out integer_array_type;
                 n : in integer) is

-- ------------------------------------------------------------
--  Procedure name: Indxx
--
--  Purpose:  Sort an array of indeces indx based on the data arrin
--            using an indexed version of a heap sort. Modified from
--            Numerical Recipes indexx

--      Input Arguments:
--         arrin - array of data used for sorting
--         n     - number of entries in arrin
--
--  Output arguments:
--      indx -  Indices specifiying the order of data in arrin

-- ------------------------------------------------------------

    j, i, l, ir, indxt : integer;
    q : float;
    indxtemp : integer_array_type(0..n-1);

    indx_j, indx_j_plus_1 : integer;
    arrin_j, arrin_j_plus_1 : float;

begin

    for k in 0 .. n - 1 loop
       indxtemp(k) := k;
    end loop;

    l := n/2;
    ir := n - 1;

    while TRUE loop
         if l > 0 then
             l := l- 1;
             indxt := indxtemp(l);
             q := arrin(indxt);
         else
             indxt := indxtemp(ir);
             q := arrin(indxt);
             indxtemp(ir) := indxtemp(0);
             ir := ir - 1;
             if ir = 0 then
                 indxtemp(0) := indxt;
                 exit;       -- exit
             end if;
         end if;

         i := l;
         j := 2 * (l + 1) - 1;
```

```
            while (j <= ir) loop
                if (j < ir and then arrin(indxtemp(j)) <
                                            arrin(indxtemp(j + 1))) then
                     j := j + 1;
                end if;
                if (q < arrin(indxtemp(j))) then
                    indxtemp(i) := indxtemp(j);
                    i := j;
                    j := j + i + 1;
                else
                    j := ir + 1;
                end if;
            end loop;

        indxtemp(i) := indxt;
    end loop;
    indx := indxtemp;

end Indxx;
```

```
separate (Build_Pkg)
procedure SortBucket (elem : in kdelem_ptr_type;
                      dim : in integer) is

-- ------------------------------------------------------------
--  Procedure name:  Sortbucket
--
--  Purpose:  Sort the entries in a KD tree bucket across
--            each dimension separately
--
--  Input arguments:
--      elem - pointer to the bucket containing the entries
--             to be sorted
--      dim  - dimensionality of the sample data
--
--  Output arguments: None
-- ------------------------------------------------------------

    entryy : kdentry_ptr_type;
    epbffr : kdentry_ptr_array_type(0 .. elem.bucket.count-1);
    mnbffr : float_array_type(0 .. elem.bucket.count-1);
    inbffr : integer_array_type(0 .. elem.bucket.count-1);
    count  : integer;

begin

    count := elem.bucket.count;
    for i in 0 .. dim - 1 loop
       entryy := elem.bucket.lists_array(0);
        for j in 0 .. count-1  loop
            mnbffr(j) := entryy.mean_array(i);
            epbffr(j) := entryy;
            entryy := entryy.next_array(0);
        end loop;

        Indxx(mnbffr, inbffr, count);

        elem.bucket.lists_array(i) := epbffr(inbffr(0));
        for j in  1 .. count - 1 loop
            epbffr(inbffr(j - 1)).next_array(i) :=
                                                epbffr(inbffr(j));
        end loop;
        epbffr(inbffr(count - 1)).next_array(i) := null;
    end loop;

end SortBucket;
```

```
with Data_Struct_Pkg; use Data_Struct_Pkg;
package MergeDown_Pkg is

    -- ------------------------------------------------------------
    -- Package specification Build_Pkg contains the declarations
    -- of all subprograms which are available for reducing the K-d
    -- tree built in module BuildKDtree to the the specified number
    -- of output centroids
    -- ------------------------------------------------------------

    procedure MergeDownKDtree (tree :  in kdtree_ptr_type;
                               ncntrds : in integer);

end MergeDown_pkg;
```

```
with Destroy_Pkg; use Destroy_Pkg;
with Build_Pkg; use Build_Pkg;
package body MergeDown_Pkg is


    procedure CollapseKDnode (elem : in out kdelem_ptr_type;
                              dim  : in integer;
                              bctr : in out integer) is separate;

    procedure GetBucketStats (elem : in kdelem_ptr_type;
                              dim  : in integer;
                              mean : in out mean_array_type;
                              wvar : in out wvar_array_type) is
                                                    separate;

    function CreateKDnode return kdelem_ptr_type is separate;

    procedure SplitBucket (oldbucket : in out kdelem_ptr_type;
                           dim : in integer;
                           bctr : in  out integer;
                           bcktarray : in out kdelem_ptr_array_type;
                           bptr : in out integer;
                           mean : in out mean_array_type;
                           wvar : in out wvar_array_type) is
                                                    separate;

    procedure BalanceKDtree (tree : in kdtree_ptr_type;
                             bcktarray : in  out
                                kdelem_ptr_array_type) is separate;

    procedure AssessCandidate (elem   : in kdelem_ptr_type;
                               dim    : in integer;
                               barray : in out kdelem_ptr_array_type;
                               ncount : in out integer) is separate;

    function BucketCompare (one : Kdelem_ptr_type;
                            two : Kdelem_ptr_type)
                                       return boolean is separate;

    procedure Quicksort (bcktarray : in  out Kdelem_ptr_array_type)
                                                    is separate;


    procedure ReduceKDbucket (elem : in kdelem_ptr_type;
                              dim : in integer) is separate;

    procedure CompressKDtree (tree : in kdtree_ptr_type;
                              nmerge : in integer;
                              bcktarray : in out
                              kdelem_ptr_array_type) is separate;

    procedure MergeDownKDtree (tree : in kdtree_ptr_type;
                               ncntrds : in integer) is separate;

end MergeDown_Pkg;
```

```
separate (MergeDown_Pkg)
procedure AssessCandidate (elem : in kdelem_ptr_type; dim : in integer;
                           barray : in  out kdelem_ptr_array_type;
                           ncount : in out integer) is

-- ------------------------------------------------------------------
--  Procedure name: Assesscandidate
--
--  Purpose: Determine the pair of bucket entries which produces
--           the minimal mean distortion when merged
--
--  Input arguments:
--      elem   - pointer to the KD tree bucket under evaluation
--      barray - array of pointer to all buckets in the KD tree
--      ncount - counter initialized to zero

--  Output arguments:
--      barray - array of pointers to all buckets that can be merged
--      ncount - counter used to keep track of the the total number
--               of buckets that can be merged
-- ------------------------------------------------------------------

ientry, jentry : kdentry_ptr_type;
reduction, dotprd, diff : float;
firsttime  : integer;

begin

    firsttime := TRUEE;

    if (elem.bucket.count > 1) then
        ientry := elem.bucket.lists_array(0);
        for i in 0 .. elem.bucket.count - 1 loop
            jentry := ientry.next_array(0);
            for j in i + 1 .. elem.bucket.count - 1 loop
                dotprd := 0.0;
                for k in 0 .. dim - 1 loop
                    diff := ientry.mean_array(k) -  jentry.mean_array(k);
                    dotprd := dotprd + diff * diff;
                end loop;
                reduction := (dotprd * ientry.weight * jentry.weight) /
                             (ientry.weight + jentry.weight);

                if ((reduction < elem.bucket.distort) or
                    (firsttime = TRUEE)) then
                    elem.bucket.distort := reduction;
                    elem.bucket.entrya := ientry;
                    elem.bucket.entryb := jentry;
                    firsttime := FALSEE;
                end if;
                jentry := jentry.next_array(0);
            end loop;   -- end j loop
            ientry := ientry.next_array(0);
        end loop;       -- end i loop
        barray(ncount) := elem;
        ncount := ncount + 1;
```

```
    end if;
end AssessCandidate;
```

```
separate (MergeDown_Pkg)
procedure BalanceKDtree (tree : in kdtree_ptr_type;
                         bcktarray : in  out kdelem_ptr_array_type) is

-- ------------------------------------------------------------------
--  Procedure name: BalanceKDtree

--  Purpose: Redistribute the entries in a KD tree so that each bucket
--           has approximately the same number of entries

--  Input arguments:
--      tree - pointer to a KD tree
--
--  Output arguments:
--      bcktarray - array used to retain a pointer to each bucket after
--                  balancing

-- ------------------------------------------------------------------

bptr  : integer;
mean  : mean_array_type(0 .. dim - 1) := (others => 0.0);
wvar  : wvar_array_type(0 .. dim - 1) := (others => 0.0);

begin

      if (tree.root.typee = KDNODEE) then
          CollapseKDnode(tree.root, tree.dim, tree.nbuckets);
      end if;

      bcktarray(0) := tree.root;
      bptr := 1;

     SplitBucket(tree.root, tree.dim, tree.nbuckets, bcktarray,
                 bptr, mean, wvar);

end BalanceKDtree;
```

```
separate (MergeDown_Pkg)
function BucketCompare (one : Kdelem_ptr_type; two : kdelem_ptr_type)
                                                    return boolean is

begin
   if one.bucket.distort < two.bucket.distort then
       return FALSE;
   else
       return TRUE;
   end if;
end BucketCompare;
```

```
separate (MergeDown_Pkg)
procedure CollapseKDnode (elem : in out kdelem_ptr_type;
                          dim : in integer;
                          bctr : in out integer) is

    FIRSTIME : BOOLEAN := TRUE;
    ptr,entptr, lentry, rentry, testentry: kdentry_ptr_type;
    bucket : kdelem_ptr_type;

begin

    if elem.node.lower.typee /= KDBUCKETT then
        CollapseKDnode (elem.node.lower, dim, bctr);
    end if;

    if elem.node.upper.typee /= KDBUCKETT then
          CollapseKDnode (elem.node.upper, dim, bctr);
    end if;

      bucket := elem.node.lower;

     for i in 0 .. dim - 1 loop
          entptr := elem.node.lower.bucket.lists_array(i);
         lentry := elem.node.lower.bucket.lists_array(i);
         rentry := elem.node.upper.bucket.lists_array(i);

         testentry := elem.node.upper.bucket.lists_array(1);

         while (lentry /= null) and  (rentry /= null) loop
             if lentry.mean_array(i) < rentry.mean_array(i) then

                 if (FIRSTIME = TRUE) then
                     elem.node.lower.bucket.lists_array(i) := lentry;
                     entptr := lentry;
                     FIRSTIME := FALSE;
                 else           -- FIRSTIME = FALSE
                     entptr.next_array(i) := lentry;
                     entptr := lentry;
                 end if;
                 lentry := lentry.next_array(i);

             else   -- case where lentry.mean_array(i) >=
                                                  rentry.mean_array(i)


                 if (FIRSTIME = TRUE) then
                     elem.node.lower.bucket.lists_array(i) := rentry;
                     entptr := rentry;
                     FIRSTIME := FALSE;
                 else
                    entptr.next_array(i) := rentry;
                    entptr := rentry;
                 end if;
                 rentry := rentry.next_array(i);

             end if;
```

```
            end loop;

            if (lentry /= null) then
                while (lentry /= null) loop
                    entptr.next_array(i) := lentry;
                    entptr := lentry;
                    lentry := lentry.next_array(i);
                end loop;
            end if;

           if (rentry /=  null) then
               while (rentry /= null) loop
                   entptr.next_array(i) := rentry;
                   entptr := rentry;
                   rentry := rentry.next_array(i);
               end loop;
           end if;
           FIRSTIME := TRUE;
        end loop;


        bucket.bucket.count := elem.node.lower.bucket.count +
                               elem.node.upper.bucket.count;

        DestroyKDbucket (elem.node.upper);
        DestroyKDnode (elem);

        bctr := bctr - 1;
        elem := bucket;

    end CollapseKDnode;
```

```
separate (MergeDown_Pkg)
procedure CompressKDtree (tree : in kdtree_ptr_type; nmerge : in integer;
                          bcktarray : in out kdelem_ptr_array_type) is

-- ------------------------------------------------------------------
--
--  Procedure name: CompressKDtree
--
--  Purpose: Procedure used to merge bucket entry pairs into single bucket
--           entries for a fixed fraction of the total number of buckets
--
--  Input arguments:
--       tree      -  pointer to a KD tree undergoing the merge
--       nmerge    -  number of bucket pairs to merge
--       bcktarray -  array of pointers to all buckets in the tree
--
--  Output arguments:
--       none
--
-- ------------------------------------------------------------------

   ptr : kdentry_ptr_type;
   ncount : integer;
   TEST_PTR : kdentry_ptr_type;
   nmerge_temp : integer := nmerge;

begin

    ncount := 0;

    --  for each bucket, one at a time, find the candidate pair,
    --  ENTRYA and ENTRYB, which has the least weighted square
    --  error (DISTORT) --  between them

    for i in 0 .. tree.nbuckets - 1 loop
        AssessCandidate(bcktarray(i), tree.dim, bcktarray, ncount);
    end loop;

    if (nmerge > ncount) then
        nmerge_temp := ncount;
    end if;

    if (ncount > 1) then
        Quicksort(bcktarray(0 .. ncount-1));
    end if;

    for i in 0 .. nmerge_temp - 1 loop
        ReduceKDbucket(bcktarray(i), tree.dim);
    end loop;

    tree.nentries := tree.nentries - nmerge_temp;

end CompressKDtree;
```

```
separate (MergeDown_Pkg)
function CreateKDnode  return kdelem_ptr_type is

-- ------------------------------------------------------------------
--
--  Function name: CreateKDnode
--
--  Purpose: Create a KD tree node
--
--  Input arguments:
--       none
--
--  Returns:
--       pointer to the newly created KD tree node
--
-- ------------------------------------------------------------------
    node : kdelem_ptr_type;

begin
    node := new kdelem(kdnode_type);
    node.typee := KDNODEE;
    return node;

end CreateKDnode;
```

```
separate (Mergedown_Pkg)
procedure GetBucketStats (elem : in kdelem_ptr_type;
                          dim  : in integer;
                          mean : in out mean_array_type;
                          wvar : in out wvar_array_type) is

    entryy  : kdentry_ptr_type;
    wgtsum : float;

begin

    wgtsum := 0.0;

    for i in 0 .. dim - 1 loop
        mean(i) := 0.0;
        wvar(i) := 0.0;
    end loop;

    entryy := elem.bucket.lists_array(0);
    while (entryy /= null) loop
        wgtsum := wgtsum + entryy.weight;
        for i in 0 .. dim - 1 loop
            mean(i) := mean(i) + entryy.wmean_array(i);
            wvar(i) := wvar(i) + entryy.wsqmn_array(i);
        end loop;

        entryy := entryy.next_array(0);
    end loop;

    for i in 0 .. dim -1 loop
        mean(i) := mean(i) / wgtsum;
        wvar(i) := (wvar(i) / wgtsum) - (mean(i) * mean(i));
    end loop;

end GetBucketStats;
```

```
separate (Mergedown_Pkg)
procedure MergeDownKDtree (tree : in kdtree_ptr_type;
                           ncntrds : in integer) is

-- ------------------------------------------------------------
--  Procedure name:  MergeDownKDtree

--  Purpose:  Reduce a KD tree to a single bucket having ncntrds
--            entries using the PNN algorithm

--  Input arguments:
--       tree - pointer to a KD tree
--       ncntrds - desired number of entries after merging
--
--  Output arguments:
--       None
--
--  Returns:
--       Nothing

-- ------------------------------------------------------------

    ptr : kdentry_ptr_type;
    nmerge, ntile : integer;
    maxbkts : integer := tree.nentries /(BUCKETSIZE / 2);
    bcktarray : kdelem_ptr_array_type(0 .. maxbkts);

begin
    while (tree.nentries > ncntrds) loop

        BalanceKDtree (tree, bcktarray);

        if KDMERGE * float(tree.nbuckets) > 1.0 then
            -- truncate by the integer conversion by subtracting .5
            ntile := integer((KDMERGE * float(tree.nbuckets)) - 0.5);

        else
            ntile := 1;
        end if;

        if (tree.nentries - ncntrds) < ntile then
            nmerge := tree.nentries - ncntrds;
        else
            nmerge := ntile;
        end if;
        CompressKDtree (tree, nmerge, bcktarray);
    end loop;

    if tree.root.typee = KDNODEE then
       CollapseKDnode (tree.root, tree.dim, tree.nbuckets);
    end if;

end MergeDownKDtree;
```

```
separate (Mergedown_Pkg)
procedure Quicksort (bcktarray : in out kdelem_ptr_array_type) is

   SIZE     : INTEGER := bcktarray'length;
   FIRST    : INTEGER := bcktarray'FIRST;
   LAST     : INTEGER := bcktarray'LAST;
   P        : INTEGER;

   procedure SWAP(A, B : in out KDELEM_PTR_TYPE) is

      TEMP : KDELEM_PTR_TYPE;

   begin

      TEMP := A;
      A := B;
      B := TEMP;
   end SWAP;

   procedure PARTITION(bcktarray : in out KDELEM_PTR_ARRAY_TYPE;
                       NEWPIVOT : out INTEGER) is

      PIVOTVALUE : KDELEM_PTR_TYPE;
      FIRST : INTEGER := bcktarray'FIRST;
      LAST  : INTEGER := bcktarray'LAST;
      LOWER : INTEGER;
      UPPER : INTEGER;
      LASTOPEN : INTEGER;

   begin

      LOWER := FIRST;
      UPPER := LAST;
      PIVOTVALUE := bcktarray(FIRST);
      LASTOPEN := FIRST;

      while LOWER < UPPER loop
         while UPPER > LOWER and then
                   BucketCompare(bcktarray(upper), PIVOTVALUE) loop

            UPPER := UPPER - 1;

         end loop;

         bcktarray(LASTOPEN) := bcktarray(UPPER);
         LASTOPEN := UPPER;

         while LOWER < UPPER and then
                    BucketCompare(PIVOTVALUE,bcktarray(lower)) loop

            LOWER := LOWER + 1;

         end loop;

         bcktarray(LASTOPEN) := bcktarray(LOWER);
         LASTOPEN := LOWER;
```

```
      end loop;

      bcktarray(LASTOPEN) := PIVOTVALUE;
      NEWPIVOT := LASTOPEN;
   end PARTITION;

begin  -- QUICKSORT

   if SIZE <= 1 then
      null;
   else
      PARTITION(bcktarray,P);
      QUICKSORT(bcktarray(FIRST .. P - 1));
      QUICKSORT(bcktarray(P + 1 .. LAST));
   end if;
end Quicksort;
```

```
with Data_Struct_Pkg; use Data_Struct_Pkg;
separate (MergeDown_Pkg)
procedure ReduceKDbucket (elem : in kdelem_ptr_type;
                          dim : in integer) is

-- ------------------------------------------------------------
--
--  Procedure name: ReduceKDbucket
--
--  Purpose: Merges a pair of bucket entries into a single entry
--
--  Input arguments:
--      elem - pointer to the KD tree bucket whose entries are to
--             be merged
--      dim  - dimension of the data within the entries
--
--  Output arguments:
--      None
-- ------------------------------------------------------------

last_entry, ientry, jentry, leptr, oldptr : kdentry_ptr_type;
newweight : float;
rmcnt : integer;
REMOVE_FIRST_ENTRY : BOOLEAN := TRUE;
FIRST_ENTRY : BOOLEAN := TRUE;
ptr, TEST_PTR : kdentry_ptr_type;

begin

   ientry := elem.bucket.entrya;
   jentry := elem.bucket.entryb;

   -- remove ientry and jentry from the list
   for i in 0 .. dim - 1 loop
       leptr := elem.bucket.lists_array(i);
       rmcnt := 0;
       while rmcnt < 2 loop
           if ((leptr /= ientry) and (leptr /= jentry))  then
               last_entry := leptr;
               leptr := leptr.next_array(i);
               REMOVE_FIRST_ENTRY := FALSE;
           else
              if REMOVE_FIRST_ENTRY = TRUE then
                 elem.bucket.lists_array(i) := leptr.next_array(i);
              else
                 last_entry.next_array(i) := leptr.next_array(i);
              end if;
              leptr := leptr.next_array(i);
              rmcnt := rmcnt + 1;
           end if;
       end loop;
       REMOVE_FIRST_ENTRY := TRUE;
   end loop;

   newweight := ientry.weight + jentry.weight;
```

```
 for i in 0 .. dim - 1 loop
    ientry.mean_array(i) := (ientry.mean_array(i) *
                             ientry.weight + jentry.mean_array(i)
                             * jentry.weight) / newweight;

    ientry.wmean_array(i) := ientry.mean_array(i) * newweight;

    ientry.wsqmn_array(i) := ientry.mean_array(i) *
                                          ientry.wmean_array(i);
 end loop;

 ientry.weight := newweight;

-- Reinsert ientry into the list in the proper order

 for i in 0 .. dim - 1 loop
     leptr := elem.bucket.lists_array(i);
     -- traverse entries that will remain unchanged in
     -- original list
     while (leptr /= null) and then
               (leptr.mean_array(i) < ientry.mean_array(i)) loop

         oldptr := leptr;
         leptr := leptr.next_array(i);
         FIRST_ENTRY := FALSE;
     end loop;

     -- insert ientry

     if (FIRST_ENTRY = TRUE) then
        elem.bucket.lists_array(i) := ientry;
     else
        oldptr.next_array(i) := ientry;
     end if;

     ientry.next_array(i) := leptr;
     FIRST_ENTRY := TRUE;
 end loop;

 elem.bucket.count := elem.bucket.count - 1;

 DestroyKDentry(jentry);

end ReduceKDbucket;
```

```
with Build_Pkg; use Build_Pkg; -- to "see" CreateKDbucket
separate (MergeDown_Pkg)
procedure SplitBucket (oldbucket : in out kdelem_ptr_type;
                       dim : in integer;
                       bctr : in out integer;
                       bcktarray : in  out kdelem_ptr_array_type;
                       bptr : in out integer;
                       mean : in out mean_array_type;
                       wvar : in out wvar_array_type) is
-- ------------------------------------------------------------
--
--  Procedure name:  Splitbucket
--
--  Purpose:  Recursive procedure used to split a bucket into two
--            smaller buckets having half as many entries
------------------------------------------------------------------

   j, bcount, medindx : integer;
   newnode : kdelem_ptr_type;
   newbucket : kdelem_ptr_type;
   dummy : kdelem_ptr_type;
   oldptr, newptr : kdentry_ptr_type;
   entryy : kdentry_ptr_type;
   FIRSTIME_LEFT : BOOLEAN := TRUE;
   FIRSTIME_RIGHT : BOOLEAN := TRUE;

begin

   if (oldbucket.bucket.count > BUCKETSIZE) then

        GetBucketStats (oldbucket, dim, mean, wvar);

       -- find dimenson with largest variance
       j := 0;
       for i in 1 .. dim - 1 loop
           if (wvar(i) > wvar(j)) then
               j := i;           -- j is dimenson with largest variance
           end if;
       end loop;

       bcount := oldbucket.bucket.count;  -- bcount is BUCKETCOUNT

       medindx := (bcount + 1) / 2;       -- median index

       newnode := CreateKDnode;

       newnode.node.dindx := j;  -- dimension to be split

       newbucket := CreateKDbucket(dim);

       newnode.node.lower := oldbucket;

       newnode.node.upper := newbucket;

       bctr := bctr + 1;   -- increment number of buckets in tree
```

```
bcktarray(bptr) :=  newbucket;

bptr := bptr + 1;    -- increment array index of bcktarray

--  traverse the entries below the median value in the
--  j dimension (the dimension with the largest variance)
--  and set entryy.splitleft = TRUE

entryy := oldbucket.bucket.lists_array(j);
for i in 0 .. medindx - 1 loop
     entryy.splitleft := TRUEE;
     entryy := entryy.next_array(j);
end loop;

--  traverse the entries above the median value in the
--  j dimension (the dimension with the largest variance)
--  and set entryy.splitleft = FALSE

for i in medindx .. bcount - 1 loop
    entryy.splitleft := FALSEE;
    entryy := entryy.next_array(j);
end loop;

oldbucket.bucket.count := medindx;

newbucket.bucket.count := bcount - medindx;

for i in 0 .. dim - 1 loop
    oldptr  :=  oldbucket.bucket.lists_array(i);
    newptr  :=  newbucket.bucket.lists_array(i);
    entryy := oldbucket.bucket.lists_array(i);

    while (entryy /= null) loop
           if (entryy.splitleft = TRUEE and
                                  FIRSTIME_LEFT = TRUE) then
               oldbucket.bucket.lists_array(i) := entryy;
               oldptr := entryy;
               FIRSTIME_LEFT := FALSE;

           elsif (entryy.splitleft = TRUEE and
                                 FIRSTIME_LEFT = FALSE) then
               oldptr.next_array(i) := entryy;
               oldptr := entryy;

           elsif (entryy.splitleft=FALSEE and
                                   FIRSTIME_RIGHT=TRUE) then
                newbucket.bucket.lists_array(i) := entryy;
                newptr := entryy;
                FIRSTIME_RIGHT := FALSE;

           elsif (entryy.splitleft=FALSEE and
                                  FIRSTIME_RIGHT=FALSE) then
                newptr.next_array(i) := entryy;
                newptr := entryy;
           end if;
```

```
                    entryy := entryy.next_array(i);

                end loop;

                FIRSTIME_LEFT := TRUE;
                FIRSTIME_RIGHT := TRUE;

                oldptr.next_array(i) := null;
                newptr.next_array(i) := null;
            end loop;


            SplitBucket(newnode.node.lower, dim, bctr,
                        bcktarray, bptr, mean, wvar);

            SplitBucket(newnode.node.upper, dim, bctr,
                        bcktarray, bptr, mean, wvar);

            oldbucket := newnode;

        end if;

    end SplitBucket;
```

```
with Data_Struct_Pkg; use Data_Struct_Pkg;
with UNCHECKED_DEALLOCATION;
package Destroy_Pkg is

    -- ------------------------------------------------------------
    -- This package specification contains the dynamic deallocation
    -- routines which are visible are to be visible
    -- ------------------------------------------------------------
    procedure DestroyKDnode (node : in out Kdelem_ptr_type);

    procedure DestroyKDtree (tree : in out Kdtree_ptr_type);

    procedure DestroyKDbucket (bucket : in out kdelem_ptr_type);

    procedure DestroyKDentry (entryy : in out kdentry_ptr_type);

end Destroy_Pkg;
```

```
package body Destroy_Pkg is

   -- ------------------------------------------------------------
   --  This package body contains the routines for deallocated
   --  memory which was previously dynamically allocated.
   -- ------------------------------------------------------------

   procedure DestroyKDnode (node : in out kdelem_ptr_type) is
                                                          separate;

   procedure DestroyKDentry (entryy : in out kdentry_ptr_type) is
                                                          separate;

   procedure DestroyKDbucket (bucket : in out kdelem_ptr_type) is
                                                          separate;

   procedure DestroyLastBucket (bucket : in out kdelem_ptr_type) is
                                                          separate;

   procedure DestroyKDtree (tree : in out kdtree_ptr_type) is
                                                          separate;

end Destroy_Pkg;
```

```
separate (Destroy_Pkg)
procedure DestroyKDbucket (bucket : in out Kdelem_ptr_type) is

-- ------------------------------------------------------------------
--
--  Procedure name: DestroyKDbucket

--  Purpose: Destroy a Kd tree bucket

--  Input arguments:
--       bucket - bucket to be destroyed
--
--  Output arguments:
--       bucket - bucket is null upon successful execution
--                                                    of FREE(bucket)

-- ------------------------------------------------------------------

   procedure FREE is new UNCHECKED_DEALLOCATION(kdelem,
                                                kdelem_ptr_type);

begin

   FREE(bucket);

end DestroyKDbucket;
```

```
separate (Destroy_Pkg)
procedure DestroyKDentry (entryy : in out Kdentry_ptr_type) is

-- ------------------------------------------------------------
--  Procedure name: DestroyKDentry

--  Purpose: Destroy a KD tree bucket entryy
--
--  Input arguments:
--      entryy - pointer to the entryy to be destroyed
--
--  Output arguments:
--      entryy - points to null upon successful execution of
--                                                    FREE(entryy)

-- ------------------------------------------------------------

procedure FREE is new UNCHECKED_DEALLOCATION (Kdentry,
                                              Kdentry_ptr_type);

begin

   FREE(entryy);

end DestroyKDentry;
```

```
separate (Destroy_Pkg)
procedure DestroyLastBucket (bucket : in out Kdelem_ptr_type) is

-- ------------------------------------------------------------
--
--  Procedure name : DestroyLastBucket
--
--  Purpose : Destroy the last bucket in a tree
--
--  Input arguments:
--     bucket - bucket to be destroyed
--
--  Output arguments:
--     None

-- ------------------------------------------------------------

   entryy, next : kdentry_ptr_type;

begin

   entryy := bucket.bucket.lists_array(0);
   while entryy /= null loop
      next := entryy.next_array(0);
      DestroyKDentry (entryy);
      entryy := next;
   end loop;

   DestroyKDbucket(bucket);

end DestroyLastBucket;
```

```
separate (Destroy_Pkg)
procedure DestroyKDnode (node : in out Kdelem_ptr_type) is

-- ------------------------------------------------------------
--  Procedure name:  DestroyKDnode
--
--  Purpose:
--      Deallocate a KD tree node
--
--  Input arguments:
--      node - pointer to the entry to be destroyed
--
--  Output arguments:
--      node - point to null upon successful execution of FREE(node)
--
-- ------------------------------------------------------------

   procedure FREE is new UNCHECKED_DEALLOCATION(kdelem,
                                                kdelem_ptr_type);

begin

   FREE(node);

end DestroyKDnode;
```

```
separate (Destroy_Pkg)
procedure DestroyKDtree (tree : in out kdtree_ptr_type) is

-- ------------------------------------------------------------------
--
--  Procedure name: DestroyKDtree
--
--  Purpose: Deallocate a KD tree structure
--
--  Input arguments:
--      tree - pointer to the KD tree
--
--  Output arguments:
--      tree - pointer to null upon successful completion of
--             FREE(tree)

-- ------------------------------------------------------------------

    procedure FREE is new UNCHECKED_DEALLOCATION(kdtree,
                                                 kdtree_ptr_type);

begin

    if tree.root /= null then
        DestroyLastBucket(tree.root);
    end if;

    FREE(tree);

end DestroyKDtree;
```

```
package timer is

-- ------------------------------------------------------------------
-- See bottom for comments on how to use the timer
-- ------------------------------------------------------------------

   type microsec_timer is private;

   procedure init_timer( obj : out microsec_timer );

   function elapsed_time( obj : in microsec_timer ) return integer;

   function identity( arg : integer ) return integer;

   function always_true return boolean;

private
   type microsec_timer is new integer;

end timer;

-- Calling sequence: Call the subprograms in this order:
--
--            loop_count : constant integer := 100; -- say...
--            :
--            dummy_timer : timer.microsec_timer;
--            dummy_elapsed_time : integer;
--            dummy_arg : integer;
--            my_timer : timer.microsec_timer;
--            my_elapsed_time : integer;
--            :
--            begin
--              :
--
--              now loop through dummy loop to determine the length
--              of time it takes to run through the loop itself.
--              We will subtract this out later.
--
--              timer.init_timer( dummy_timer );
--              for i in 1..loop_count
--              loop
--                dummy_arg := timer.identity( dummy_arg );
--              end loop;
--              if timer.always_true
--              then
--                dummy_elapsed_time := timer.elapsed_time( dummy_timer );
--              end if;
--
--            -- now run through the same loop and add the actual code that
--            -- you want to time.
--
--              timer.init_timer( my_timer );
--              for i in 1..loop_count
--              loop
--              -- do something here that needs timing...
--              timer.identity( dummy_arg );
```

```
--              end loop;
--              if timer.always_true
--              then
--                my_elapsed_time := timer.elapsed_time( my_timer );
--              end if;
--
--           -- now subtract off the dummy loop time and divide by the
--           -- number of times that the loop occurs.  This equals one
--           -- iteration of the interesting code.
--
--              time_for_one_iteration :=
--               ( my_elapsed_time - dummy_elapsed_time ) / loop_count );

-- note that the value of my_timer is not changed by a call to
-- timer.elapsed_time.  You can have as many timers as you wish.
```

```
with system ; use system ;
with condition_handling ; use condition_handling ;
with starlet ; use starlet ;

package body timer is

-- ------------------------------------------------------------
--
-- Package: timer
-- -------
-- Description: This package provides two subprograms which are called
-- to time operations in the vms environment.
-- -------
-- References: Please see the VAX/VMS System Services Reference Manual
-- under the $GETJPI system service description.
-- -------
-- -------
-- Revision history:  S. French 25-Jan-1989
-- -------
-- Host processor: VAX/VMS 4.7
-- -------
-- Package dependencies:  This package interfaces with the VMS system
-- service $GETJPI to get the cpu time.
-- ------------------------------------------------------------------

   function cpu_time_clock return integer is
     cputim : integer ;
     pragma volatile ( cputim ) ;
     jpi_status : cond_value_type ;
     jpi_item_list : constant item_list_type :=
                     ( ( 4 , jpi_cputim , cputim'address , address_zero ) ,
                       ( 0 , 0 , address_zero , address_zero ) ) ;

   begin
     -- call getjpi to set cputim to total accumulated cpu time
     -- (in millisecond tics)

     getjpi ( status => jpi_status , itmlst => jpi_item_list ) ;
     return (cputim * 10);
   end cpu_time_clock ;

   procedure init_timer( obj : out microsec_timer ) is
   begin
     obj := microsec_timer( cpu_time_clock );
   end init_timer;

   function elapsed_time( obj : in microsec_timer ) return integer is
   begin
     return cpu_time_clock - integer( obj );
   end elapsed_time;

   function identity( arg : integer ) return integer is
     some_value : integer := 0;
   begin
     some_value := some_value + arg;
     return some_value;
```

```
   end identity;

   function always_true return boolean is
     bool_value : boolean := true;
   begin
     return bool_value;
   end always_true;

begin
  null;
end timer;
```

```
package timer is

-- ------------------------------------------------------------------
--  See bottom for comments on how to use the timer
-- ------------------------------------------------------------------

   type microsec_timer is private;

   procedure init_timer( obj : out microsec timer );

   function elapsed_time( obj : in microsec_timer ) return integer;

   function identity( arg : integer ) return integer;

   function always_true return boolean;

private
   type microsec_timer is new integer;

end timer;

-- ------------------------------------------------------------------
--
-- Package: timer
--
-- Description: This package provides two subprograms which are called
-- to time operations.
-- -----
-- Calling sequence: Call the subprograms in this order:
--
--     loop_count : constant integer := 100; -- say...
--     dummy_timer : timer.microsec_timer;
--     dummy_elapsed_time : integer;
--     dummy_arg : integer;
--     my_timer : timer.microsec_timer;
--     my_elapsed_time : integer;
--
--  begin
--
--     now loop through dummy loop to determine the length of time
--     it takes to run through the loop itself. We will subtract
--     this out later.
--
--     timer.init_timer( dummy_timer );
--     for i in 1..loop_count loop
--        dummy_arg := timer.identity( dummy_arg );
--     end loop;
--
--     if timer.always_true then
--        dummy_elapsed_time := timer.elapsed_time( dummy_timer );
--     end if;
--
--     now run through the same loop and add the actual code that
--     you want to time.
--
--     timer.init_timer( my_timer );
```

```
--     for i in 1..loop_count loop
--        do something here that needs timing...
--        timer.identity( dummy_arg );
--     end loop;

--     if timer.always_true then
--        my_elapsed_time := timer.elapsed_time( my_timer );
--     end if;
--
--     now subtract off the dummy loop time and divide by the
--     number of times that the loop occurs.  This equals one
--     iteration of the interesting code.
--
--     time_for_one_iteration :=
--         ( my_elapsed_time - dummy_elapsed_time ) / loop_count );

--   note that the value of my_timer is not changed by a call to
--   timer.elapsed_time.  You can have as many timers as you wish.
--   Revision history:  S. French 25-Jan-1989
--
```

```
package body timer is

-- ------------------------------------------------------------------
--
-- Package: timer
-- -------
-- Description: This package provides two subprograms which are called
-- to time functions in the unix environment. This timer was used to
-- perform timing on the MIPS Magnum 3000 computer system.
-- -------
-- References: Please see the man page clock for information about how
-- the unix time is obtained.
-- -------
-- Revision history:  S. French 25-Jan-1989
-- -------
-- Host processor: gppe gppe 3_10 UMIPS mips m120-5 ATT_V3_0
-- -------
-- Package dependencies:  This package interfaces with the UNIX system
-- call "clock".
-- ------------------------------------------------------------------

   function unix_clock return integer;
       pragma INTERFACE(C, unix_clock);
       pragma INTERFACE_NAME(unix_clock,"clock");

   procedure init_timer( obj : out microsec_timer ) is
   begin
     obj := microsec_timer( unix_clock );
   end init_timer;

   function elapsed_time( obj : in microsec_timer ) return integer is
   begin
     return unix_clock - integer( obj );
   end elapsed_time;

   function identity( arg : integer ) return integer is
     some_value : integer := 0;
   begin
     some_value := some_value + arg;
     return some_value;
   end identity;

   function always_true return boolean is
     bool_value : boolean := true;
   begin
     return bool_value;
   end always_true;

begin
  null;
end timer;
```

```
with text_io; use text_io;
with data_struct_pkg; use data_struct_pkg;
procedure READ_DATA (INFILE : IN OUT FILE_TYPE;
                     positions : out means_array_type;
                     weights   : out weights_array_type) is

 -- ------------------------------------------------------------
 --
 --  Procedure Name:  READ_DATA
 --
 --  Purpose:  Procedure READ_DATA is called by Procedure Main
 --            to read in the input data file.
 -- ------------------------------------------------------------
   i : integer := 0;
   j : integer := 0;
   package FLOAT_IO is new TEXT_IO.FLOAT_IO(FLOAT);

begin

      while not END_OF_FILE(INFILE) loop
        FLOAT_IO.GET(INFILE, positions(i));
        i := i + 1;
        FLOAT_IO.GET(INFILE, positions(i));
        i := i + 1;
        FLOAT_IO.GET(INFILE, weights(j));
        j := j + 1;
        SKIP_LINE(INFILE);
    end loop;

    CLOSE(INFILE);

end READ_DATA;
```

58.000000 19.000000 62.000000
59.000000 19.000000 74.000000
60.000000 19.000000 70.000000
57.000000 20.000000 76.000000
58.000000 20.000000 90.000000
59.000000 20.000000 91.000000
60.000000 20.000000 78.000000
61.000000 20.000000 52.000000
56.000000 21.000000 75.000000
57.000000 21.000000 96.000000
58.000000 21.000000 101.000000
59.000000 21.000000 91.000000
60.000000 21.000000 67.000000
56.000000 22.000000 75.000000
57.000000 22.000000 100.000000
58.000000 22.000000 108.000000
59.000000 22.000000 104.000000
60.000000 22.000000 92.000000
61.000000 22.000000 72.000000
56.000000 23.000000 84.000000
57.000000 23.000000 135.000000
58.000000 23.000000 159.000000
59.000000 23.000000 165.000000
60.000000 23.000000 155.000000
61.000000 23.000000 124.000000
62.000000 23.000000 61.000000
56.000000 24.000000 105.000000
57.000000 24.000000 164.000000
58.000000 24.000000 190.000000
59.000000 24.000000 196.000000
60.000000 24.000000 185.000000
61.000000 24.000000 151.000000
62.000000 24.000000 74.000000
55.000000 25.000000 59.000000
56.000000 25.000000 109.000000
57.000000 25.000000 173.000000
58.000000 25.000000 200.000000
59.000000 25.000000 206.000000
60.000000 25.000000 194.000000
61.000000 25.000000 160.000000
62.000000 25.000000 81.000000
63.000000 25.000000 58.000000
36.000000 26.000000 52.000000
56.000000 26.000000 105.000000
57.000000 26.000000 165.000000
58.000000 26.000000 191.000000
59.000000 26.000000 197.000000
60.000000 26.000000 185.000000
61.000000 26.000000 152.000000
62.000000 26.000000 80.000000
63.000000 26.000000 58.000000
36.000000 27.000000 60.000000
37.000000 27.000000 73.000000
40.000000 27.000000 85.000000

U.S. Government Printing Office:1992—648-127/62109